INFORMIX®

Advanced INFORMIX-4GL Development

Advanced INFORMIX-4GL Development

Part No. 502-5-224-1-999999-2
Version 12-93

Advanced INFORMIX-4GL Development

Acknowledgments

Course Developer ..... Cindy Lee Dowling
Course Production Coordinator ..... Judy Perry
Course Development Manager ..... Sharon Poore
Production Assistance ..... Pam Lucia
Chris McAdams
Director, Education Services ..... Karen Rathjens
Technical Review Team ..... Pete Carpino
Art DeYong, Ken Dowling, Andrew Hulme, Dawn Maneval,
Dennis Pimple, Sandy Sloan, Nancy Suffron

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without the prior written consent of:

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

Informix, HyperScript, Wingz, C-ISAM, and Perform are registered trademarks and OpenCase, OpenCase/ToolBus, ToolBus, DataExtract, 4GL++, 4GL/GX, ACE, ViewPoint, and SuperView are trademarks of Informix Software, Inc. IBM, IBM PC, PC/XT, PC-AT, OS/2, and DRDA are registered trademarks, and PC-DOS is a trademark of International Business Machines Corporation. Microsoft and MS-DOS are registered trademarks, and Windows is a trademark of Microsoft Corporation. Ada is a registered trademark of the U.S. Government. Motif and OSF/Motif are trademarks of Open Software Foundation. NFS, Sun, and Sun Microsystems are registered trademarks and Open Windows, SunOS, and PC-NFS are trademarks of Sun Microsystems. OPEN LOOK is a registered trademarks of AT&T. UNIX is a registered trademark, licensed exclusively by X/Open Company Ltd. in the United Kingdom and other countries. X/Open is a registered trademark of X/Open Company Ltd. in the United Kingdom and other countries. PostScript is a registered trademark of Adobe Systems, Inc. dBase is a registered trademark of Borland International, Inc. NetWare is a registered trademark of Novell, Inc. INFORMIX-4GL Forms and INFORMIX-Menus are licensed to Informix Software, Inc. by FourGen Software, Inc.

Printed in the United States of America.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software FAR Clause 52.227-7013 (and any other applicable license provisions set forth in the Government contract).

© 1993 by Informix Software, Inc.

Digitalized by atm.

Chapter 1 Creating a Development Environment for 4GL Development 2111 12-93

2 The Demonstration Database 2122 12-93

3 Using MAKE 2109 12-93

4 Using a Source Code Control System 2110 12-93

5 Using the Interactive Debugger 2105 12-93

6 Creating Advanced Screen Arrays 2112 12-93

7 Creating 4GL Libraries 2106 12-93

8 Calling C Functions from 4GL 2103 12-93

9 Creating Custom Runners and Debuggers 2104 12-93

10 4GL Optimization 2108 12-93

11 Locking, Transactions and Performance 2107 12-93

12 Style Guide for INFORMIX-4GL Development 2119 12-93

13 Sample Makefiles 2100 12-93

14 4GL Utilities Source Code 2102 12-93

15 Sample Generic Pick List Window Source Code 2099 12-93

Appendix A Style Guide for INFORMIX-4GL Development 2119 12-93

B Sample Makefiles 2100 12-93

C 4GL Utilities Source Code 2102 12-93

D Sample Generic Pick List Window Source Code 2099 12-93

Objectives

At the end of this course, you will be able to:

- Compile 4GL applications using MAKE
- Use SCCS for source code control
- Use the Interactive Debugger to debug 4GL applications
- Create advanced screen arrays
- Create 4GL libraries
- Implement error handling routines
- Call a C function from a 4GL program
- Call a 4GL function from a C program
- Create custom runners for 4GL RDS
- Implement 4GL optimization techniques
- Understand performance issues related to transactions and locking

Prerequisites

To maximize the benefits of this course, we require that you have met the following prerequisites:

- Experience developing INFORMIX-4GL applications
- *Developing Applications Using INFORMIX-4GL* or equivalent knowledge

If this course is being delivered at a customer site, we require that the following software be available for use during the class:

- INFORMIX-4GL C Compiler Version
- INFORMIX-4GL Rapid Development System
- INFORMIX-4GL Interactive Debugger
- MAKE and SCCS

ther Information

If you would like to learn more about INFORMIX-4GL, we recommend these resources for further reading:

- *INFORMIX-4GL Interactive Debugger*
- *INFORMIX-4GL by Example*

To order product documentation, or to obtain further information regarding Informix training, please call **(800)438-7627.**

Comments

Thank you for attending this training class. If you have any suggestions regarding the course materials, we would like to hear from you. Please contact our Education Services department at **(415)926-6300.**

Chapter

1

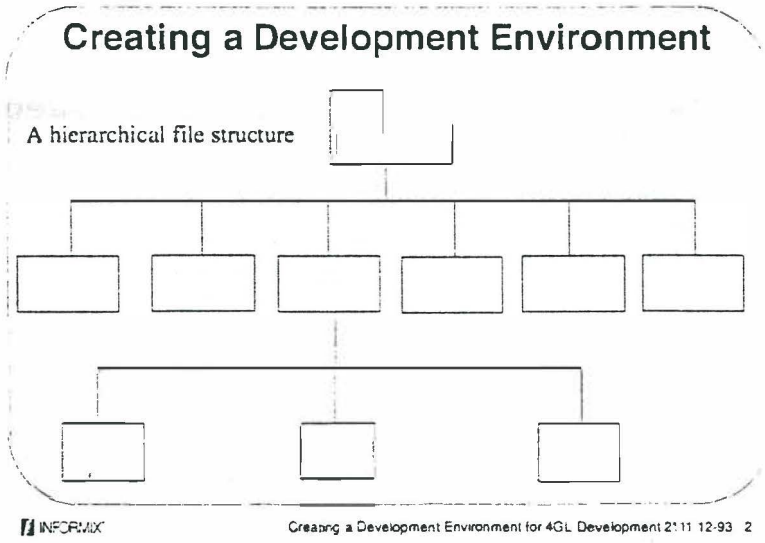
Creating a Development Environment for 4GL Development

Objectives

At the end of this module, you will be able to:

- Explain the benefits of a structured development environment
- Describe the development directory tree
- Describe the SCCS directory tree
- Understand the benefits of stub programming
- Understand client and server code structure
- Use the **arg_val( )** and **num_args( )** library functions
- Use the **fgl_getenv( )** library function
- Explain the role of a configuration manager

© 1993 Informix Software, Inc.


Creating a Development Environment
A hierarchical file structure
INFORMIX
Creating a Development Environment for 4GL Development 2.11 12-93 2

Creating a Development Environment

A hierarchical file structure implemented in the development environment:

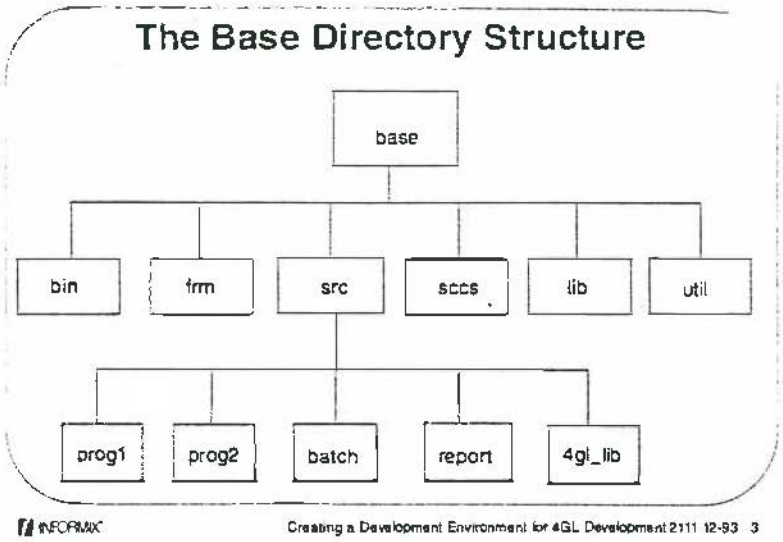
- creates a manageable but flexible environment
- supports many developers
- simplifies application integration and porting
- provides control structures for multiple releases
- leverages the functionality of MAKE and SCCS
- supports multiple environments for development, testing and production

To control the development environment and provide an area for each developer to work, the development environment is divided into smaller work areas. The work areas need to be centrally located to provide for ease of control. The central location assists the quality control process by defining a specific repository for programs.

A development environment must provide a work area for the developer and a repository for SCCS files. The environment must also support other functionality such as a common repository of libraries which can be linked into programs.

The Key is Consistency!

This course will present one approach to creating a productive development environment. You may select a different approach for your needs. The key is ***consistency***!



The Base Directory Structure

The top-level directory is your base development ($BASEDIR) directory. Your base directory should be placed in a central location that can be accessed by all developers.

A brief description of the contents of each directory is listed below:

bin	Application executables, binary files. Developers should include this in their path.
frm	Compiled forms.
src	Source files for the applications. This directory is divided into sub-directories.
sccs	SCCS s-files. This directory will have the same sub-directory structure as the source (src) directory.
lib	Compiled library functions that are shared by programs.
util or **etc**	Utilities and tools used in the development process, including custom runners. Developers should include this in their path.

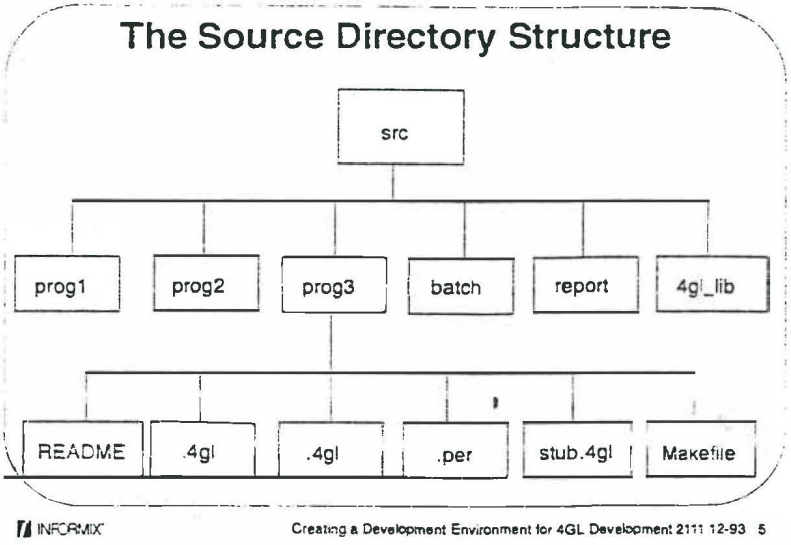
If you are using the C compiled and RDS version in development, you may want to separate the bin directory as follows:

4gi_bin	Contains all .4gi executables generated through RDS.

4ge_bin	Contains all **.4ge** executables generated through **c4gl**.

You may require other support directories in your development environment. For example:

obj	This directory is divided into sub-directories for the placement of **.4go**, **.o** and **.frm** files.
incl	Header files, optional files used in C development.
doc	Application documentation.



The Source Directory Structure

The source directory can be organized according to the number of developers and the size of the project. A project with more than one developer should have source separated into one directory per program.

The 4gl_lib

This directory contains common 4GL modules and global files which pertain to all modules within an application. Common 4GL modules can be compiled and archived into a library that each program can link into its executable. 4GL libraries are discussed in detail in a later module.

Other directories may include:

c_lib	This directory contains all C and ESQL/C source code which pertain to all modules within an application.
p_lib	This directory contains p-code which pertains to all modules within an application.
per_lib	This directory contains all **.per** files that are generic to all modules in an application.

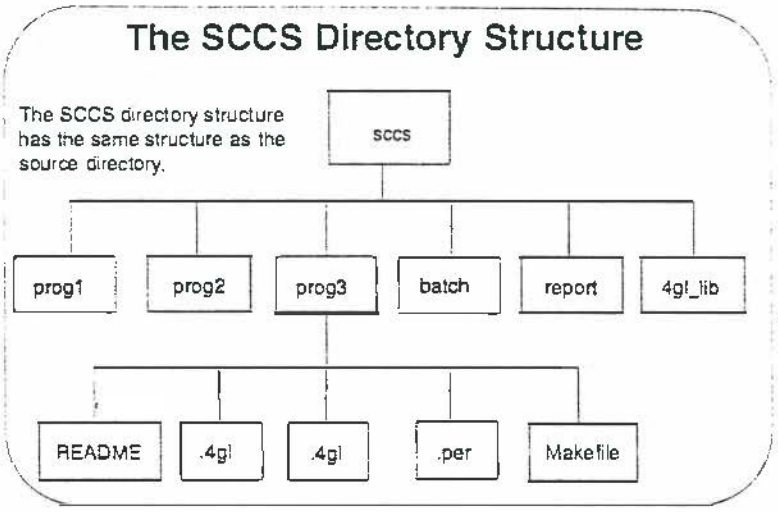
The Source Sub-Directory

A brief description of the contents of each working sub-directory under the source directory is listed below:

README	Contains a brief description of the directory, and any other pertinent information.

.4gl	4GL source files.
.per	4GL form source files.
Makefile	Makefile contains instructions on how to compile a runable executable with the files located in the directory.
stub	4GL stub program used for unit testing.

To increase productivity, each sub-directory should be able to compile a runable executable. This provides for unit testing of individual pieces of the application with minimal impact on the other developers. This will be accomplished by the use of a 4GL *stub* program.



The SCCS Directory Structure

One approach from the software development standpoint is to make the source directory structure and the SCCS directory structure the same. The duplicate structures keep the 4GL source and SCCS files separate and the development directories clean. The split also facilitates the management of the SCCS files. SCCS supports the development process, but the SCCS files are not needed in the development directories.

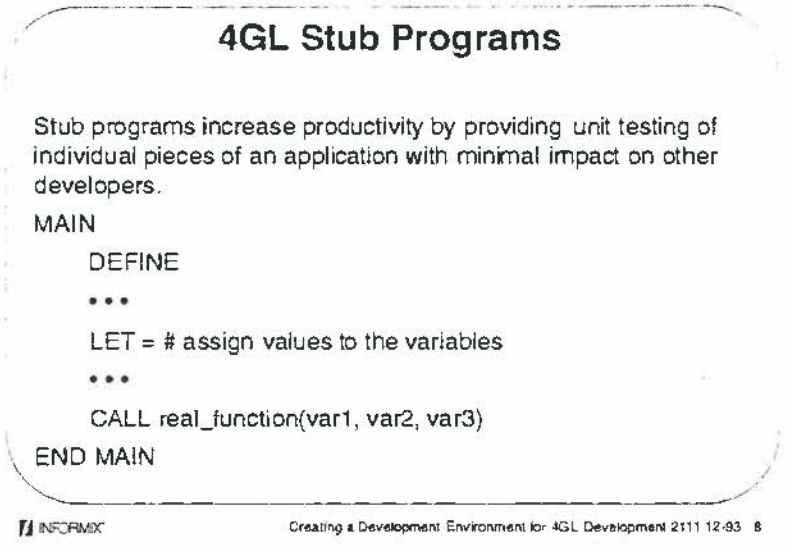
INFORMIX-4GL development directories can become cluttered with p-code and c-code objects. By having the SCCS files in a separate directory structure elsewhere, the number of files in the working directory is limited.

Another benefit of splitting the directory tree is to ease in porting the application. The directory tree can be duplicated and the proper files extracted from SCCS. The duplicate tree can be moved to a new platform and compiled.

The two-tree environment, a source directory tree and an SCCS directory tree, is one approach. You may choose a different approach to the SCCS directory tree structure. *The key is consistency!*

SCCS will be discussed in detail in a later module.

Tip: All ASCII files that you edit and change can be placed under SCCS. This includes form specification files (.per) and Makefiles.


4GL Stub Programs
Stub programs increase productivity by providing unit testing of individual pieces of an application with minimal impact on other developers.
MAIN
DEFINE
• • •
LET = # assign values to the variables
• • •
CALL real_function(var1, var2, var3)
END MAIN
INFORMIX
Creating a Development Environment for 4GL Development 2111 12-93 8

4GL Stub Programs

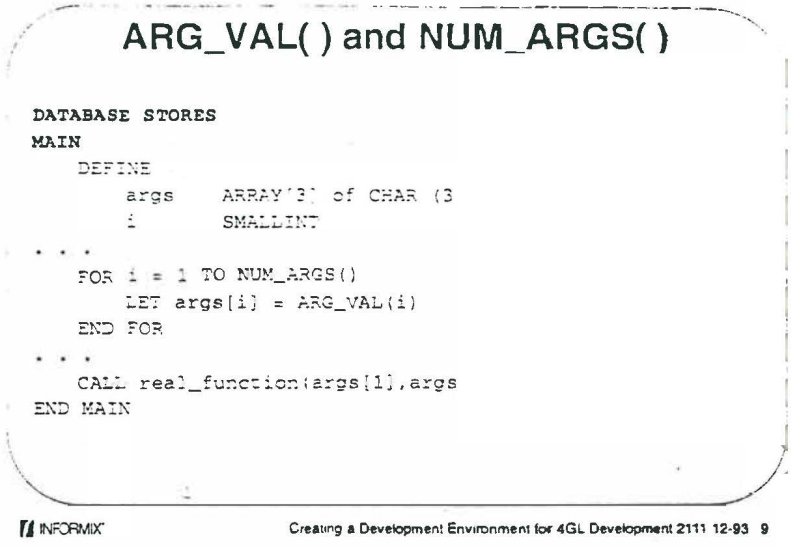
Each sub-directory under the source directory should be able to compile a runable executable. This provides for unit testing of individual pieces of the application with minimal impact on the other developers. Stub development needs to be accomplished knowing that *the end goal is integration into a larger executable.*

No Global File

There is only one globals file per executable, so a globals file must be established for the larger executable. Do not create a globals file for each directory unless the directory is the final production application directory.

The stub programs that are developed must contain a **MAIN** and set any variables required by the *"real"* function. The setting of these variables can be accomplished in different ways:

- You can pass values to the program from the command line. Command line arguments are processed with the built-in INFORMIX-4GL functions, NUM_ARGS( ) and ARG_VAL( ). This is the preferred method used throughout this course.
- You can open a stub form and run an INPUT statement. After the user enters information in the stub form, then call the *"real"* functions with the values.
- You can use LET statements and hard code the variable values. This is a poor method because it requires the program to be re-compiled to test other data sets.



ARG_VAL( ) and NUM_ARGS( )

ARG_VAL( ) and NUM_ARGS( ) are INFORMIX-4GL library functions that you can include in your 4GL source code. The 4GL compiler recognizes the name of the library function and automatically includes the function in your executable.

Design the stub program to allow arguments to be passed into the program from the command line using the ARG_VAL( ) and NUM_ARGS( ) library functions. The ARG_VAL( ) function retrieves individual arguments during program execution. The NUM_ARGS( ) function determines how many arguments were passed in at the command line.

Command Line Arguments

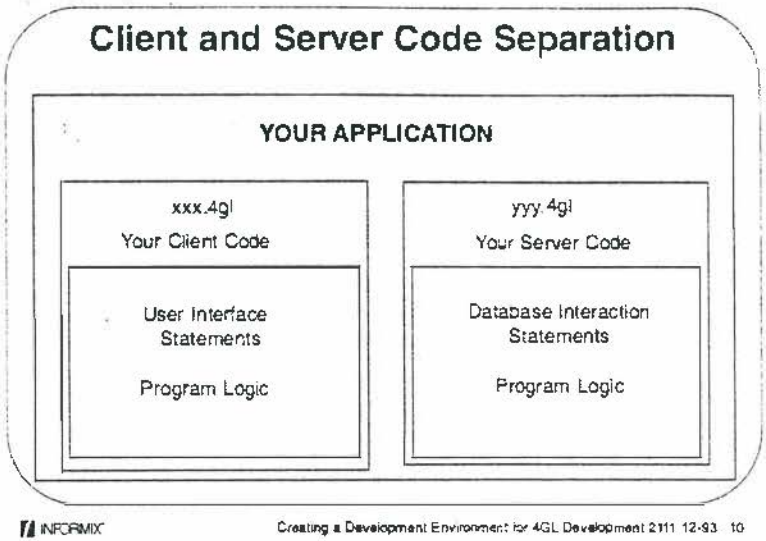
Suppose your stub program can accept three manufacturer codes as command line arguments. The command line would be:

```
% stub.4ge ANZ HRO SMT (INFORMIX-4GL C Compiler Version)
% fglgo stub.4gi ANZ HRO SMT (INFORMIX-4GL RDS Version)
```

The NUM_ARGS( ) function returns the value 3. Executing the LET statement in the FOR loop assigns the following values to the elements of the **args** array:

args[1] = ANZ, args[2] = HRO, args[3] = SMT

Tip: You want to test for the correct number of arguments. If the incorrect number was passed in, display a usage statement.



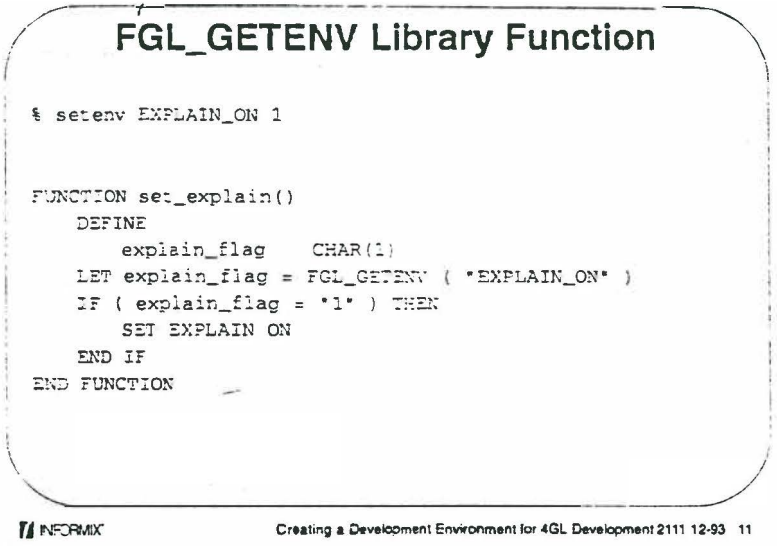
Client and Server Code Separation

This course will present a 4GL code development structure that separates database interaction statements from the rest of your 4GL application. The 4GL code is split into *client* and *server* modules as follows:

- Client modules contain screen and logic statements.
- Server modules contain only database interaction statements. Server modules do not interact with the user, only with the database server.

Structuring your application in client and server functions isolates your SQL statements and provides many benefits:

- Separating client and server functions promotes well-structured, functionalized code.
- Database access statements are isolated from user interface statements which enhances maintainability.
- Stripping out the user interface statements provides an easy path for future migration to other front ends such as a graphical user interface.
- Separating client and server functions into separate modules is *required* for a distributed transaction programming environment (using INFORMIX-TP/Toolkit and INFORMIX-4GL).

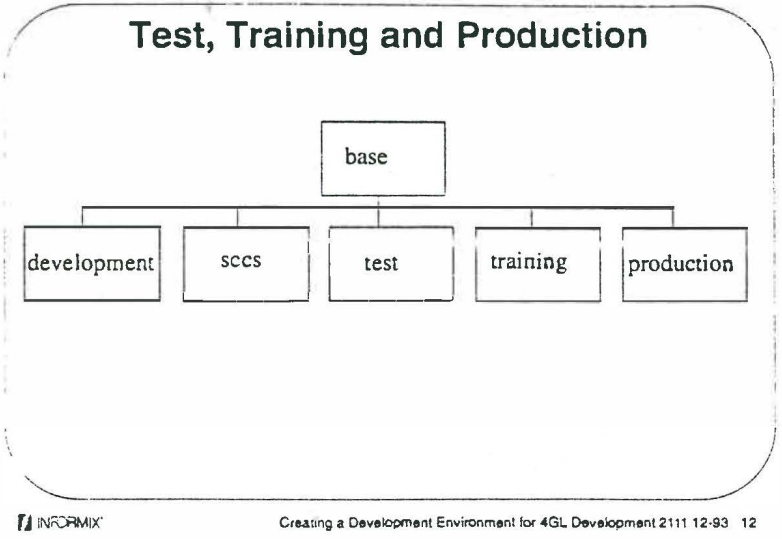


FGL_GETENV Library Function

The FGL_GETENV( ) library function returns the character value of an environment variable. This is a very powerful library function as it allows you to examine the values of environment variables anywhere within a 4GL program.

It can be especially valuable during the development phase to implement various functions and tools that are used during the development process but not used in production.

Note: The FGL_GETENV( ) library function is available with INFORMIX-4GL Version 4.1.



Test, Training and Production

We have discussed the concept of a two-tree environment, a *base* (development) directory tree and an SCCS duplicate directory tree. Projects may also require separate environments for testing, training and production. These directories, like the SCCS directory, should generally have similar structures.

The Test Directory

The test directory can be used by QA for testing purposes. The use of stub programs to perform unit testing can greatly reduce the testing life cycle.

The Training Directory

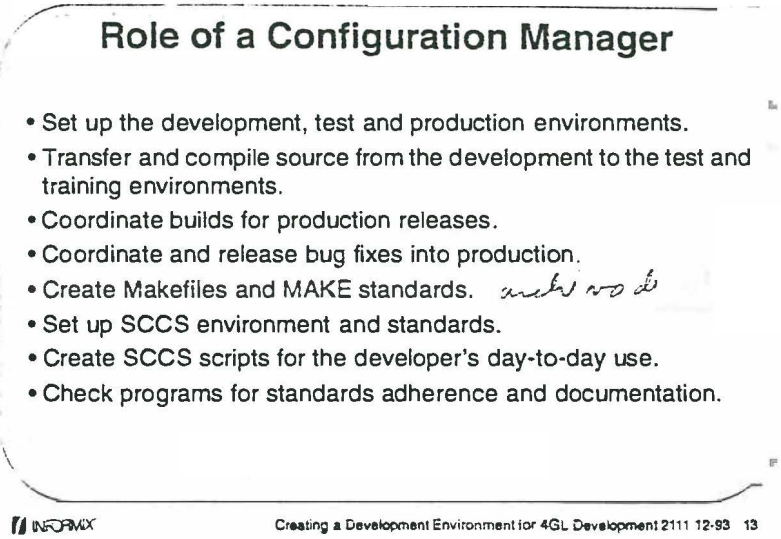
Most projects require some type of environment for user training. This may be a separate machine or a separate directory environment on the development machine.

The Production Directory

All development projects should have a production directory. The directory should hold copies of production libraries. Keeping copies of the production libraries prevents having to re-create the libraries if a fix is needed to production. You can use SCCS to re-create the production libraries, but consider these two factors:

- If a fix is needed to production, usually this must be accomplished as expediently as possible. Re-creating the libraries could be very time consuming.
- You run the risk of introducing new errors when re-creating the libraries.

Creating libraries will be covered in detail in a later module.

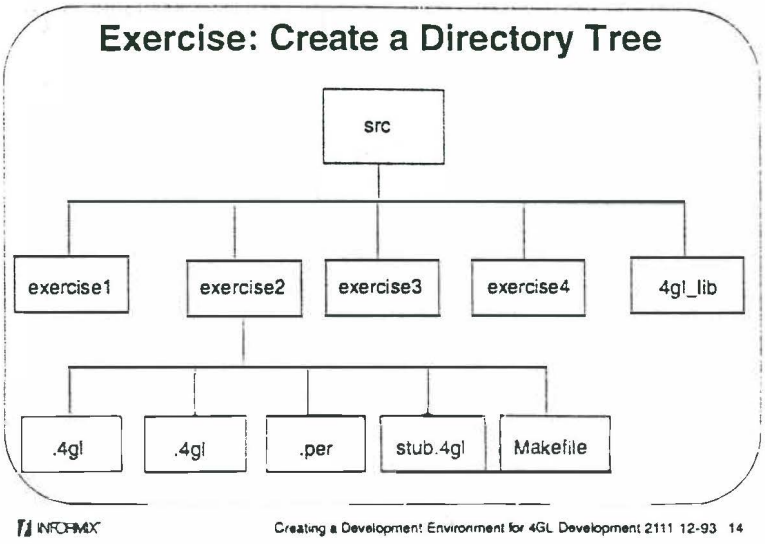

Role of a Configuration Manager
• Set up the development, test and production environments.
• Transfer and compile source from the development to the test and training environments.
• Coordinate builds for production releases.
• Coordinate and release bug fixes into production.
• Create Makefiles and MAKE standards.
• Set up SCCS environment and standards.
• Create SCCS scripts for the developer's day-to-day use.
• Check programs for standards adherence and documentation.
INFORMIX
Creating a Development Environment for 4GL Development 2111 12-93 13

Role of a Configuration Manager

In a development environment with many programmers, checks must be in place to ensure that standards and guidelines are being followed. Many organizations designate a Configuration Manager to develop, coordinate and monitor the development environment standards.

The Configuration Manager is the key individual responsible for creating and maintaining a productive development environment. The Configuration Manager may be a designated person (in a large project) or a role of one of the lead developers (in a smaller project).

The Configuration Manager should know MAKE, SCCS, and how to write shell scripts. One goal is to remove the complexity of MAKE and SCCS and other aspects of the development environment from the day-to-day operations of the applications programmer.



Exercise: Create a Directory Tree

Use the UNIX mkdir *dir_name* command to create a source directory tree as displayed above. This directory tree will be used throughout the course.

Step 1

You must obtain the base directory from your instructor. It is recommended to set an environment variable called **BASEDIR** to reference your base directory path.

Step 2

Create a source directory structure with five sub-directories: **exercise1**, **exercise2**, exercise3, exercise4 and 4gl_lib.

Step 3

Create an SCCS directory at the same level of the source (**src**) directory with one sub-directory, **exercise1**.

You will use the **exercise1**, **exercise2**, **exercise3** and **exercise4** sub-directories under your source directory to develop the exercises throughout this course. The **4gl_lib** directory will be used for common 4GL modules archived into a library in a later module. The **sccs/exercise1** directory will be used to store SCCS files in a later module.

Each working sub-directory will eventually contain your:

- 4GL source files (.4gl)
- 4GL form files (.per)
- 4GL stub file (stub.4gl)
- a Makefile

Solutions

Solution to Exercise: Create a Directory Tree

Step 1

```
setenv BASEDIR /dir_name/. . .
```

Change to your base directory.

```
cd $BASEDIR
```

Step 2

```
mkdir src
cd src
mkdir exercise1 exercise2 exercise3 exercise4 4gl_lib
cd ..
```

Step 3

```
mkdir sccs
cd sccs
mkdir exercise1
```

Chapter

2

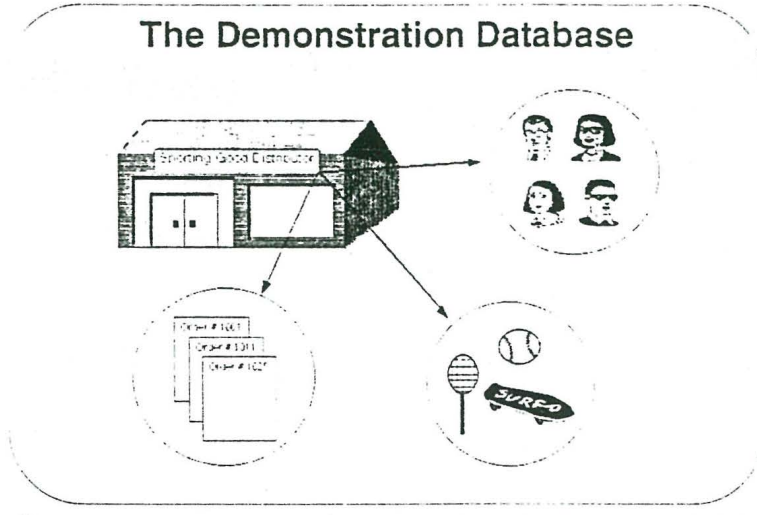
The Demonstration Database

Objectives

At the end of this module, you will be able to:

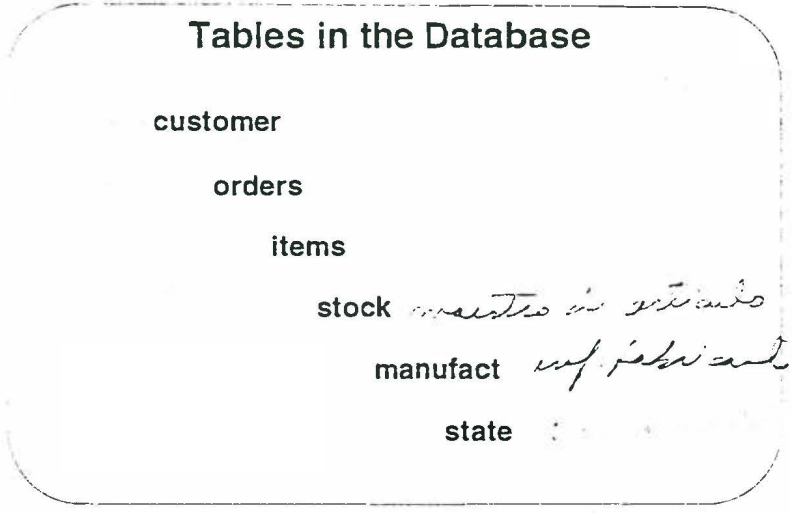
- Describe the schema of the demonstration database
- Understand the relationship between the tables
- Create a copy of the demonstration database

© 1993 Informix Software, Inc.



The Demonstration Database

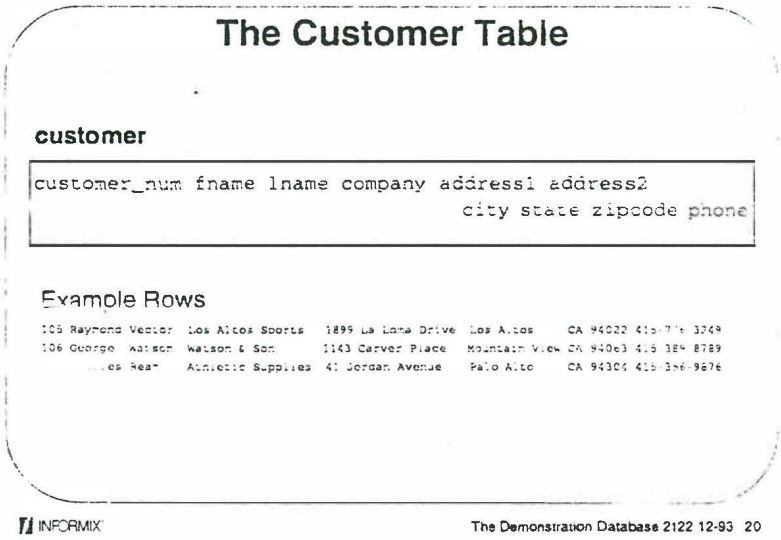
The **stores** database contains information about a fictitious wholesale sporting goods distributor. The example forms, reports and command files that make up a demonstration application are also included. You can use this material to follow the examples in the product manuals or to practice with INFORMIX-4GL.



Tables in the Database

The stores database contains information about a fictitious wholesale sporting goods distributor that services stores in the United States. The database includes six tables:

- customer
- orders
- items
- stock
- manufact
- state


The Customer Table
customer
customer_num fname lname company address1 address2
city state zipcode phone
Example Rows
106 Raymond Vector Los Altos Sports 1899 La Loma Drive Los Altos CA 94022
106 George Watson Watson & Son 1143 Carver Place Mountain View CA 94063 415 389 8789
Athletic Supplies 41 Jordan Avenue Palo Alto CA 94304 415 356-9876
INFORMIX
The Demonstration Database 2122 12-93 20

The customer Table

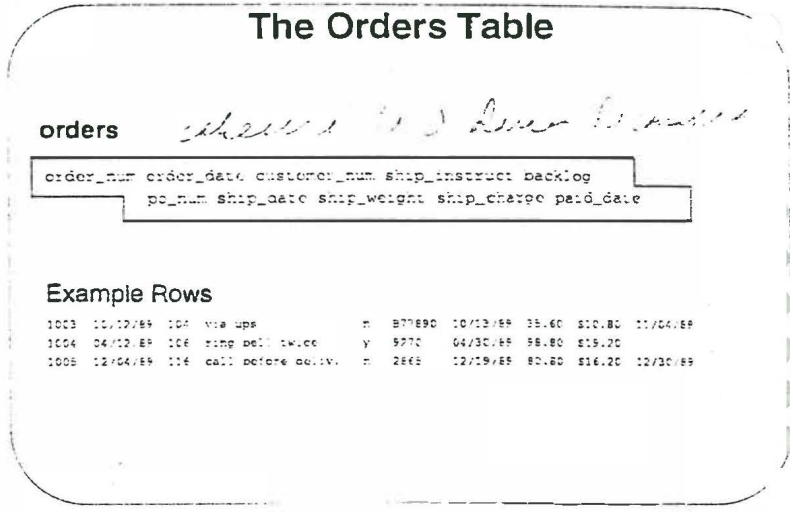
The customer table contains information about 18 stores that order sporting goods from the distributor. Customer information consists of the following columns:

- Customer account number
- Name of the company representative (two columns)
- Company name
- Address (five columns)
- Telephone number

The Data Types

Below is a listing of the columns in the **customer** table and their data types.

customer_num	serial
fname	character 15
lname	character 15
company	character 20
address1	character 20
address2	character 20
city	character 15
state	character 2
zipcode	character 5
phone	character 18


The Orders Table
orders
order_num order_date customer_num ship_instruct backlog
po_num ship_date ship_weight ship_charge paid_date
Example Rows
1003 10/12/89 104 via ups n B77890 10/13/89 35.60 $10.80 11/04/89
1004 04/12.89 106 ring bell twice y 9270 04/30/89 98.80 $19.20
1005 12/04/89 116 call before deliv. n 2865 12/19/89 80.80 $16.20 12/30/89
INFORMIX
The Demonstration Database 2122 12-93 21

The Orders Table

The **orders** table contains 15 orders placed by customers. Order information consists of the following ten columns:

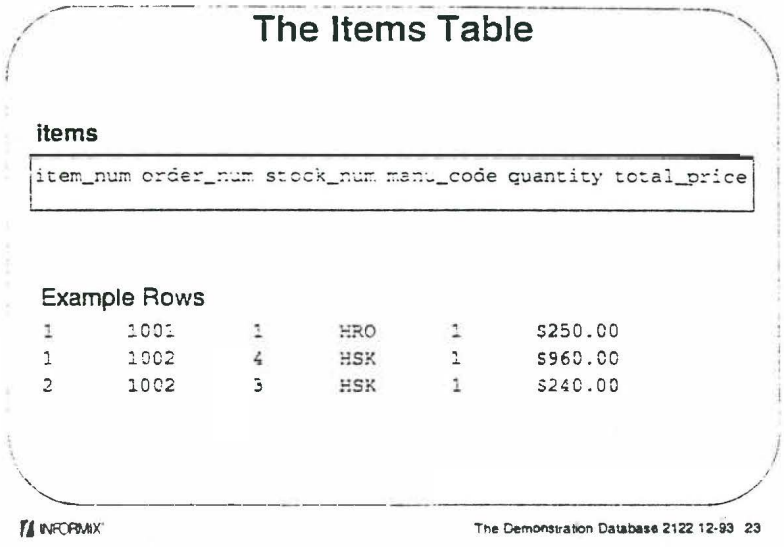
- Order number
- Date the order was placed
- Customer number
- Shipping instructions
- Whether a backlog exists
- Customer purchase order number
- Date the order was shipped
- Weight of the order
- Shipment charge
- Date the customer paid for the order

The Data Types

Below is a listing of the columns in the **orders** table and their data types.

order_num	serial
order_date	date
customer_num	integer
ship_instruct	character 40
backlog	character 1
po_num	character 10

ship_date	date
ship_weight	decimal
ship_charge	money
paid_date	date


The Items Table
items
item_num order_num stock_num manu_code quantity total_price
Example Rows
1 1001 1 HRO 1 $250.00
1 1002 4 HSK 1 $960.00
2 1002 3 HSK 1 $240.00
INFORMIX
The Demonstration Database 2122 12-93 23

The Items Table

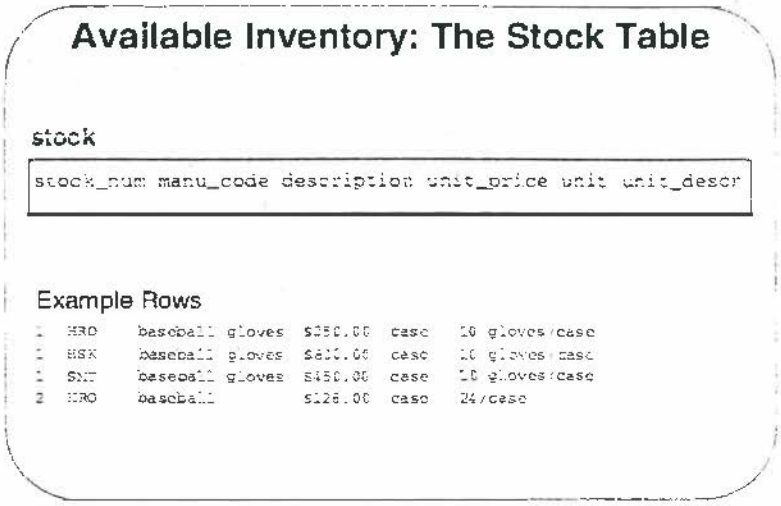
The **items** table keeps track of individual line items in each order. For example, some orders only contain one item line. Other orders contain multiple item lines. Item information consists of the following six columns:

- Line item number
- Order number
- Stock number of the item
- Manufacturer's code
- Quantity ordered
- Total price of the ordered items
  (**quantity** times unit_price from the **stock** table)

The Data Types

Below is a listing of the columns in the **items** table and their data types.

item_num	smallint
order_num	integer
stock_num	smallint
manu_code	character 3
quantity	smallint
total_price	money


Available Inventory: The Stock Table
stock
stock_num manu_code description unit_price unit unit_descr
Example Rows
1 HRO baseball gloves $250.00 case 10 gloves/case
1 HSK baseball gloves $800.00 case 10 gloves/case
1 SMT baseball gloves $450.00 case 10 gloves/case
2 HRO baseball $126.00 case 24/case
INFORMIX
The Demonstration Database 2122 12-93 24

Available Inventory: The Stock Table

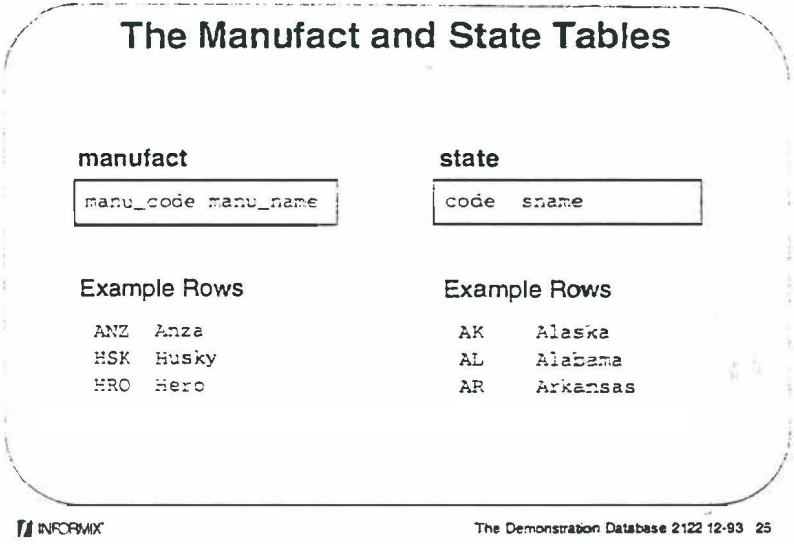
The stock table contains the articles sold by the distributor. Stock information consists of the following six columns:

- Stock number
- Manufacturer's code
- Description of the item
- Price of each unit
- Unit by which the item must be ordered
- Description of the unit

The Data Types

Below is a listing of the columns in the stock table and their data types.

stock_num	smallint
manu_code	character 3
description	character 15
unit_price	money
unit	character 4
unit_descr	character 15



The Manufact and State Tables

The manufact table contains the following information about the manufacturer:

- Manufacturer's code, a three-letter abbreviation
- The full name of the manufacturer

The state table contains the following information about each state:

- State code, a two-letter abbreviation, uppercase
- The full name of the state

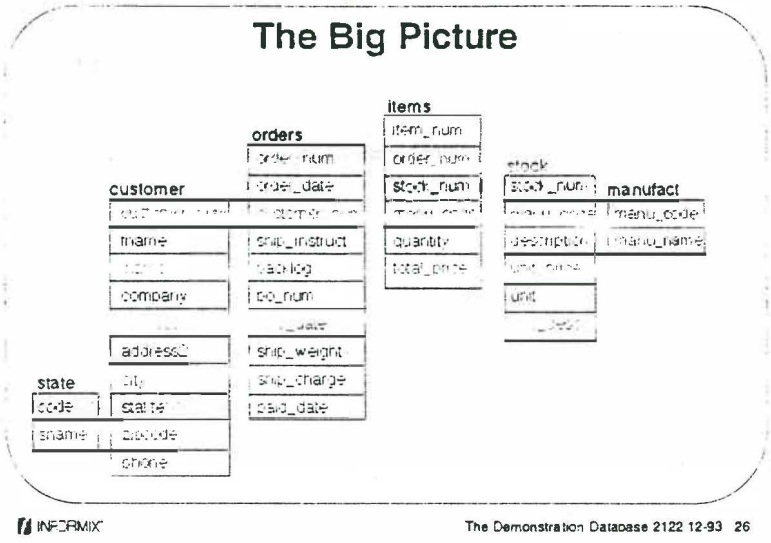
The Data Types

Below is a listing of the columns in the manufact table and their data types.

manu_code	character 3
manu_name	character 15

Below is a listing of the columns in the state table and their data types.

code	character 2
sname	character 15



The Big Picture

The ability to join data from different tables is one of the most important features of a relational database management system. Instead of storing the identical data in several tables, a relational database management system allows you to retrieve data from several tables at once and display it as if it were stored in a single table.

Joining allows you to rearrange the information of a database. This flexibility lets you create new relationships between tables without redesigning your database. You can easily expand the scope of a database by adding new tables that can be joined to existing tables.

The six tables of the **stores** database are linked together by the join columns, as indicated in the diagram above.

Example Data for the Stores Database

customer

customer_num	fname	lname	company1	address1	address2	city	state	zipcode	phone
101	Ludwig	Pauli	All Sports Supplies	213 Erstwild Court		Sunnyvale	CA	94086	408-789-8075
102	Carole	Sadler	Sports Spot	785 Geary St		San Francisco	CA	94117	415-822-1289
103	Philip	Currie	Phil's Sports	654 Poplar	P.O. Box 3498	Palo Alto	CA	94303	415-328-4543
104	Anthony	Higgins	Play Ball!	East Shopping Cntr.	422 Bay Road	Redwood City	CA	94026	415-368-1100
105	Raymond	Vector	Los Altos Sports	1899 La Loma Drive		Los Altos	CA	94022	415-776-3249
106	George	Watson	Watson & Son	1143 Carver Place		Mountain View	CA	94063	415-389-8789
107	Charles	Ream	Athletic Supplies	41 Jordan Avenue		Palo Alto	CA	94304	415-356-9876
108	Donald	Quinn	Quinn's Sports	587 Alverado		Redwood City	CA	94063	415-544-8729
109	Jane	Miller	Sport Stuff	Mayfair Mart	7345 Ross Blvd.	Sunnyvale	CA	94086	408-723-8789
110	Roy	Jaeger	AA Athletics	520 Topaz Way		Redwood City	CA	94062	415-743-3611
111	Frances	Keyes	Sports Center	3199 Sterling Court		Sunnyvale	CA	94085	408-277-7245
112	Margaret	Lawson	Runners & Others	234 Wyandotte Way		Los Altos	CA	94022	415-887-7235
113	Lana	Beatty	Sportstown	654 Oak Grove		Menlo Park	CA	94025	415-356-9982
114	Frank	Albertson	Sporting Place	947 Waverly Place		Redwood City	CA	94062	415-884-6677
115	Alfred	Grant	Gold Medal Sports	776 Gary Avenue		Menlo Park	CA	94025	415-356-1123
116	Jean	Parmelee	Olympic City	1104 Spinosa Drive		Mountain View	CA	94040	415-534-8822
117	Arnold	Sipes	Kids Korner	850 Lytton Court		Redwood City	CA	94063	415-245-4578
118	Dick	Baxter	Blue Ribbon Sports	5427 College		Oakland	CA	94609	415-655-0011

orders

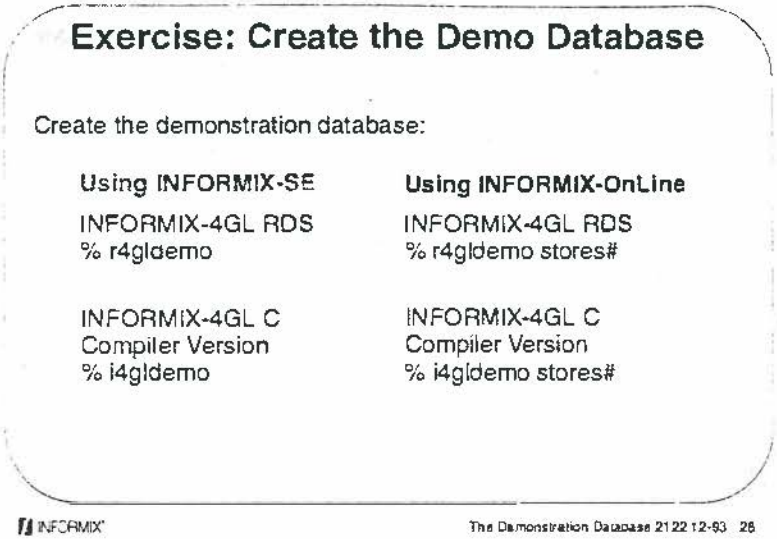
order_num	order_date	customer_num	ship_instruct	backlog	po_num	ship_date	ship_weight	ship_charge	paid_date
1001	01/20/89	104	ups	n	B77836	02/01/89	20.40	10.00	03/22/89
1002	05/01/89	101	po on box; deliver back door only	n	9270	06/06/89	50.60	15.30	07/03/89
1003	10/12/89	104	via ups	n	B77890	10/13/89	35.60	10.80	11/04/89
1004	04/12/89	106	ring bell twice	y	8006	04/30/8	98.80	19.20	
1005	12/04/89	116	call before delivering	n	2865	12/19/89	80.80	16.20	12/30/89
1006	09/19/89	112	after 10 am	y	Q13557		70.80	14.20	
1007	03/25/89	117		n	278693	04/23/89	125.90	25.20	
1008	11/17/89	110	closed Monday	y	LZ230	12/06/89	45.60	13.80	12/21/89
1009	02/14/89	111	door next to supersaver	n	4745	03/04/89	20.40	10.00	04/21/89
1010	05/29/89	115	deliver 776 Gary if no answer	n	429Q	06/08/89	40.60	12.30	07/22/89
1011	03/23/89	104	ups	n	B77897	04/13/89	10.40	5.00	06/01/89
1012	06/05/89	117		n	278701	06/09/89	70.80	14.20	
1013	09/01/89	104	via ups	n	B77930	09/17/89	60.80	12.20	10/10/89
1014	5/01/89	106	ring bell, kick door loudly	n	8052	05/10/89	40.60	12.30	07/18/89
1015	07/10/89	110	closed Mon	n	MA003	08/01/89	20.60	6.30	08/31/89

items

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1001	1	HRO	1	250.0
1	1002	4	HSK	1	960.0
2	1002	3	HSK	1	240.0
1	1003	9	ANZ	1	20.0
2	1003	8	ANZ	1	840.0
3	1003	5	ANZ	1	99.0
1	1004	1	HRO	5	960.0
2	1004	2	HRO	1	125.0
3	1004	3	HSK	1	240.0
4	1004	1	HSK	1	800.0
1	1005	1	NRG	1	280.0
2	1005	5	ANZ	10	198.0
3	1005	6	SMT	10	36.0
4	1005	6	ANZ	1	48.0

items continued

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1006	5	SMT	1	425.0
2	1006	5	NRG	5	190.0
3	1006	5	ANZ	5	99.0
4	1006	5	SMT	5	36.0
5	1006	6	ANZ	1	48.0
...					
1	1013	5	ANZ	1	19.8
2	1013	6	SMT	1	36.0
3	1013	6	ANZ	1	48.0
4	1013	9	ANZ	2	40.0
1	1014	4	HSK	1	960.0
2	1014	4	HRO	1	480.0
1	1015	1	SMT	1	450.0



Exercise: Create the Demo Database

You need to create the demonstration database for the appropriate database server and version of INFORMIX-4GL that you are using as listed above.

- In your home directory, create a sub-directory called **demo1**
- Change to the new directory
- Run the appropriate demonstration script as instructed by your instructor. Substitute the pound sign (#) with your student number.

All of the demonstration reports and command files are copied into the **demo1** directory. If you are using INFORMIX-SE, both demo scripts create a sub-directory called **stores.dbs** and places the stores database files there.

INFORMIX

Chapter

Using MAKE

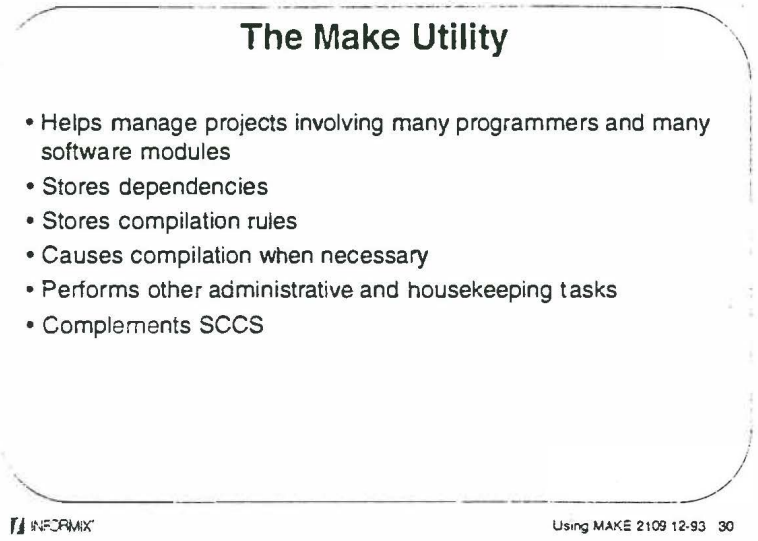
3

Objectives

At the end of this module, you will be able to:

- Describe the basic functionality of MAKE
- Explain the basic components of a description file
- Define suffixes and dependencies
- Create custom targets for 4GL development
- Use command line syntax to compile 4GL source code and forms
- Create a 4GL stub program

© 1993 Informix Software, Inc.


The Make Utility
• Helps manage projects involving many programmers and many software modules
• Stores dependencies
• Stores compilation rules
• Causes compilation when necessary
• Performs other administrative and housekeeping tasks
• Complements SCCS
INFORMIX
Using MAKE 2109 12-93 30

The Make Utility

The make utility is a UNIX utility which generates and executes UNIX shell commands. The make utility automatically keeps tracks of files that have changed and causes recompiling when necessary. In the long term, this will save in development time.

Dependencies

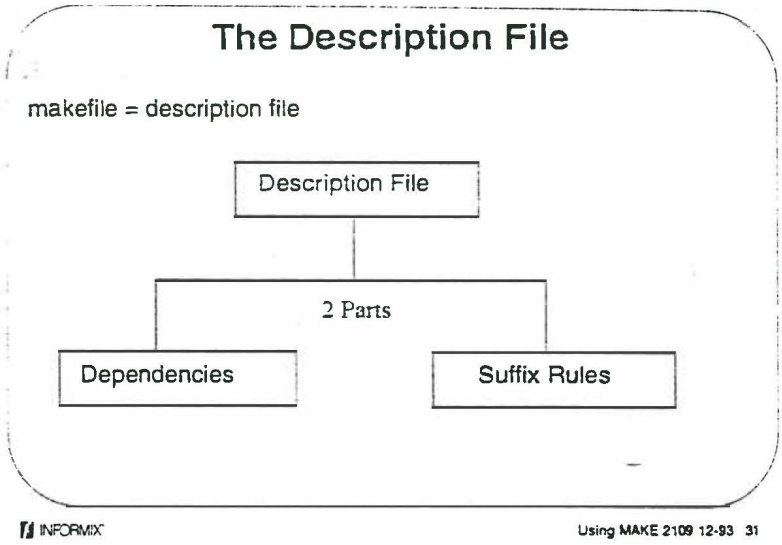
It is not uncommon for several source files to depend on another source file. The make utility determines the dependencies between files and executes the appropriate commands. An example of a dependency is that *xxx*.o depends on ***xxx*.4gl**. If ***xxx*.4gl** is newer than ***xxx*.p**, **make** will generate and execute the necessary commands to create a new *xxx*.o. By specifying this hierarchy of dependent relations, **make** will take care of recompiling the necessary files when a file is changed.

Other Administrative Tasks

The **make** utility can perform other administrative tasks such as removing old files, building libraries, and checking out files from SCCS if necessary. It is recommended to set up your makefiles to do as many of these tasks automatically as possible.

Make and SCCS

The **make** utility and SCCS work very well together. You can set up your **makefile** so that if **make** does not find a particular file, then it will automatically look for the **s-file** in your SCCS directory. If it finds the file, it will issue the get command to retrieve the file. After make is finished with the extracted SCCS file, it will issue a delta command to SCCS.



The Description File

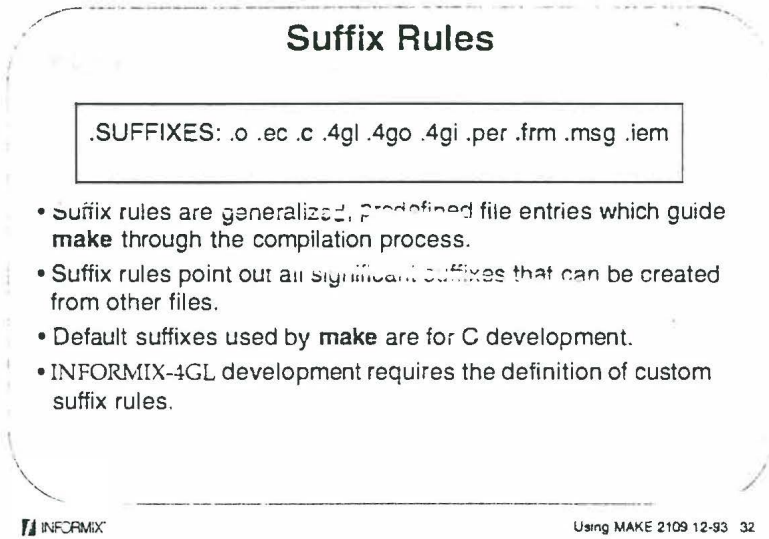
The **make** utility uses a *description file* (command file), known as the **makefile** or **Makefile**, and some general templates to determine the dependencies between the files. In addition to the description file, **make** derives necessary information from the *last-modified* time of files as recorded in the file system.

The description file consists of two different components:

- Suffix rules
- Dependencies

The suffix rules and dependencies are defined in the description file that **make** uses to compile the applications. The default name of the command file is either Makefile or makefile. If you use a different name, then you must supply the name as an argument whenever you execute the make command:

```
make -f filename
```

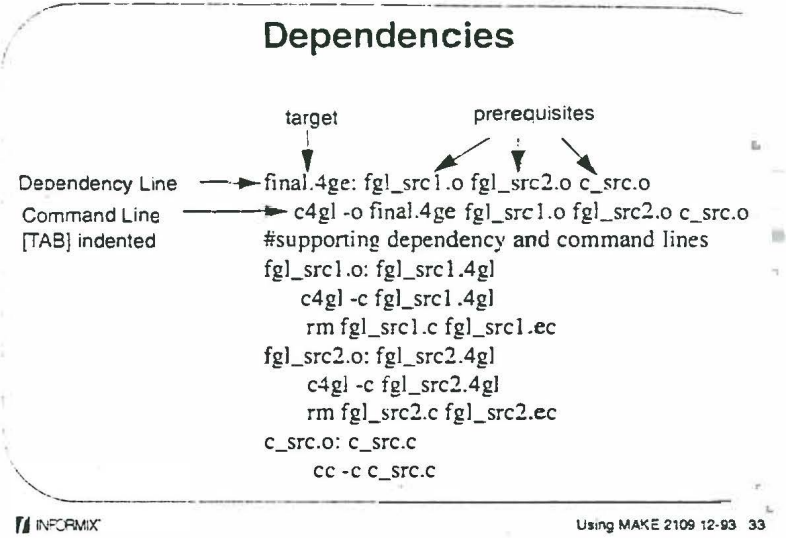

Suffix Rules
.SUFFIXES: .o .ec .c .4gl .4go .4gi .per .frm .msg .iem
• Suffix rules are generalized, predefined file entries which guide make through the compilation process.
• Suffix rules point out all significant suffixes that can be created from other files.
• Default suffixes used by make are for C development.
• INFORMIX-4GL development requires the definition of custom suffix rules.
INFORMIX
Using MAKE 2109 12-93 32

Suffix Rules

Suffix rules are generalized predefined file entries which guide **make** through the compilation process. Suffix rules point out significant suffixes for files that can be created from other files. In an INFORMIX-4GL example, suffix rules show that *xxx*.o can be created from ***xxx*.4gl**.

Custom Suffix Rules

The default suffix rules used by **make** are for C development and do not support INFORMIX-4GL development. It is not necessary to specify C suffixes; make will follow the logical suffix translation paths for C. You will need to specify custom suffix rules for INFORMIX-4GL development.



Dependencies

The make utility determines the dependencies between files and executes the appropriate commands. Dependency lines show the relationship between files and are followed by one or more command lines.

The dependency line has two parts separated by a colon (:). To the left o the colon is the *target* and to the right of the colon are the *prerequisites.* Following the dependency line will be one or more *tab* indented command lines to show how to make the target from the prerequisites.

The makefile above contains four entries. Each entry consists of a line containing a colon, the dependency line, and one or more command lines beginning with exactly one tab. To the left of the colon are the targets; to the right of the colon are the prerequisites to **make** the target

The first entry says that the target **final.4ge** depends on the prerequisites **fgl_src1.o, fgl_src2.o** and **c_src.o**. The command line associated with the target specifies how to link the object files to make the executable **final.4ge.**

The **make** utility will execute the command line if **final.4ge** does not exist, or if any of the prerequisites were modified after the last-modified date of **final.4ge**.

Note: The end-of-line is indicated on the command line by the \ character.

Command Line Syntax for c4gl

```
c4gl {[-ansi] [-e] [-a] [-otherargs] [-o outfile] source.4gl [otheresql.ec]
     [otnersrc.c] [otherobj.o] [yourlib] }
```

Example:

```
c4gl -o final.4ge module1.4gl module2.4gl
```

The **c4gl** command compiles the 4GL source code modules, **module1.4gl** and **module2.4gl** and produces an executable program called **final.4ge**. You can then run the executable **final.4ge**.

Command Line Syntax for c4gl

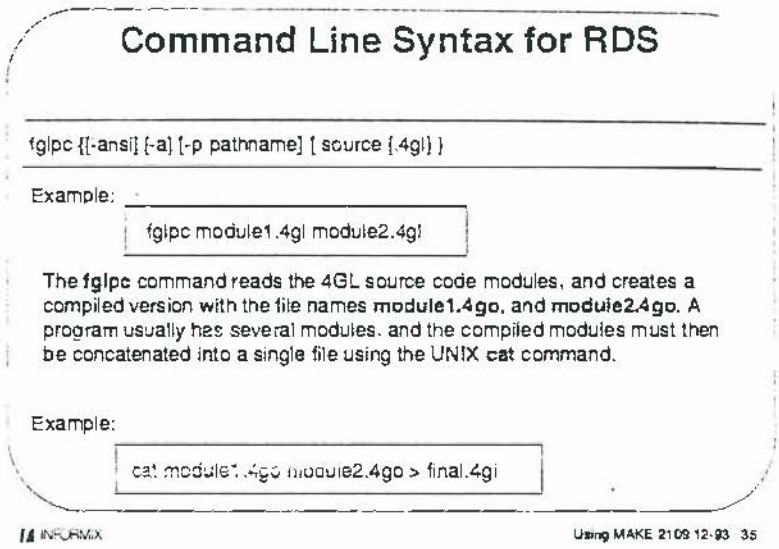
An INFORMIX-4GL program will include several source code modules. Before you can execute a 4GL program, you have to preprocess and compile all of the source modules and link them with any library files that they reference. You can do all of the steps with the c4gl command:

1. **4GL** (.4gl) source code files are preprocessed to produce INFORMIX-ESQL/C code (**.ec**).
2. INFORMIX-ESQL/C code files are preprocessed to produce C code (**.c**).
3. C code files are compiled to produce object files (**.o**).
4. Object files are linked to the INFORMIX-ESQL/C libraries and to any other libraries specified at the command line.

If compilation errors occur, **c4gl** creates an error file with a .err extension.

Note: The INFORMIX-ESQL/C source files, C source files and C object files are intermediate steps in producing the executable. Besides 4GL source files, you can include files of these types at the command line when using **c4gl**.

For more information regarding command line syntax, refer to the INFORMIX-4GL *Reference Manual*.


Command Line Syntax for RDS
fglpc {[-ansi] [-a] [-p pathname] [source [.4gl]]
Example:
fglpc module1.4gl module2.4gl
The fglpc command reads the 4GL source code modules, and creates a compiled version with the file names module1.4go, and module2.4go. A program usually has several modules, and the compiled modules must then be concatenated into a single file using the UNIX cat command.
Example:
cat module1.4go module2.4go > final.4gi
INFORMIX
Using MAKE 2109 12-93 35

Command Line Syntax for RDS

INFORMIX-4GL Rapid Development System is used to compile, run, or debug p-code from your 4GL source modules. Special RDS commands to perform the steps at the command line include:

fglpc	4GL P-Code Compiler
fglgo	4GL P-Code Runner
fgldb	4GL Interactive Debugger

To create an application with the INFORMIX-4GL Rapid Development System requires the following steps:

1. Compile the 4GL (.4gl) source code files into a p-code file (.4go).
2. Combine the multiple p-code files (.4go) into a single file (.4gi).
3. Invoke the 4GL p-code runner, specifying the 4GL program (.4gi).

fglgo

You can run the 4GL program listed above by using the 4GL p-code runner at the command line:

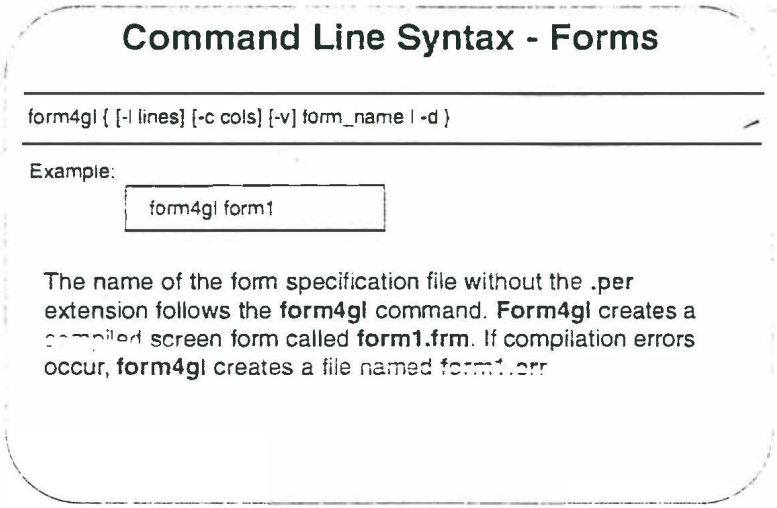
```
fglgo final
```

fgldb

You can run the debugger at the system prompt by using the command line:

```
fgldb filename
```

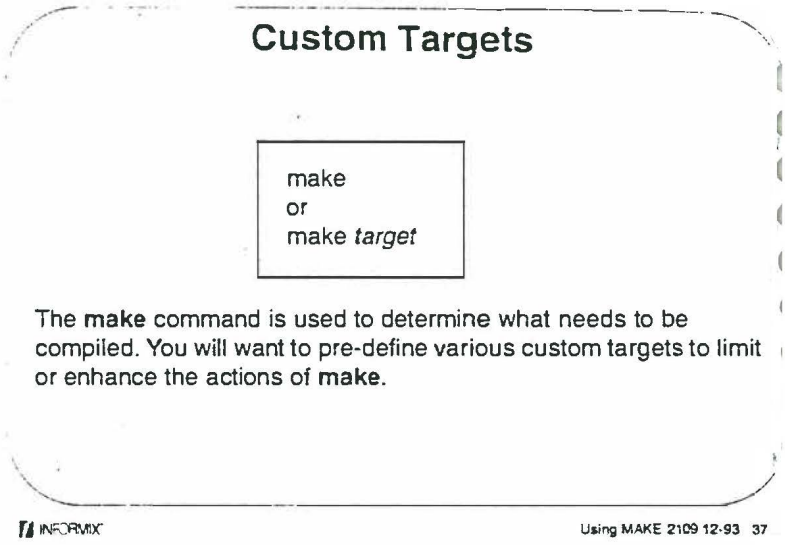
Filename is any runable 4GL file that you produced by an fglpc command.


Command Line Syntax - Forms
form4gl { [-l lines] [-c cols] [-v] form_name | -d }
Example:
form4gl form1
The name of the form specification file without the .per extension follows the form4gl command. Form4gl creates a compiled screen form called form1.frm. If compilation errors occur, form4gl creates a file named form1.err
INFORMIX
Using MAKE 2109 12-93 36

Command Line Syntax - Forms

You can use the **form4gl** command to compile form specification files at the command line.

For more information regarding command line syntax, refer to the INFORMIX-4GL *Reference Manual*.


Custom Targets
make
or
make target
The make command is used to determine what needs to be compiled. You will want to pre-define various custom targets to limit or enhance the actions of make.
INFORMIX
Using MAKE 2109 12-93 37

Custom Targets

To invoke **make**, you can issue one of the following commands:

- **make**
- **make** *target*

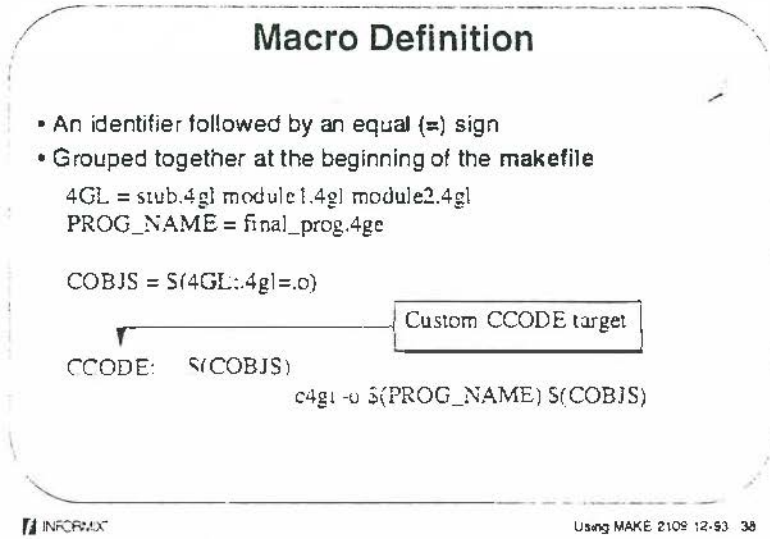
In the first option, the utility will make the first target it finds in the makefile. In the second option, you specify the target to make after the make command. More than one target can be specified on the command line. Before linking the object files, **make** checks to see if each of the prerequisites are up to date.

Suggested Targets

Following are suggested custom targets to support INFORMIX-4GL development:

make PCODE	The target PCODE will compile the INFORMIX-4GL Rapid Development System version of the application.
make CCODE	The target CCODE will compile the INFORMIX-4GL C Compiler Version of the application.
make FORM	The target FORM will compile the forms.

Other custom targets might perform CCODE and PCODE library creation.


Macro Definition
• An identifier followed by an equal (=) sign
• Grouped together at the beginning of the makefile
4GL = stub.4gl module1.4gl module2.4gl
PROG_NAME = final_prog.4ge
COBJS = $(4GL:.4gl=.o)
Custom CCODE target
CCODE: $(COBJS)
c4gl -o $(PROG_NAME) $(COBJS)
INFORMIX
Using MAKE 2109 12-93 38

Macro Definition

A *macro definition* is an identifier followed by an equal (=) sign. The identifier must not be preceded by a colon (:) or tab. The name to the left of the equal sign is assigned the string of characters following the equal sign. Blanks and tabs immediately to the left and right of the equal sign are ignored. Typically, macro definitions are grouped together at the beginning of the makefile.

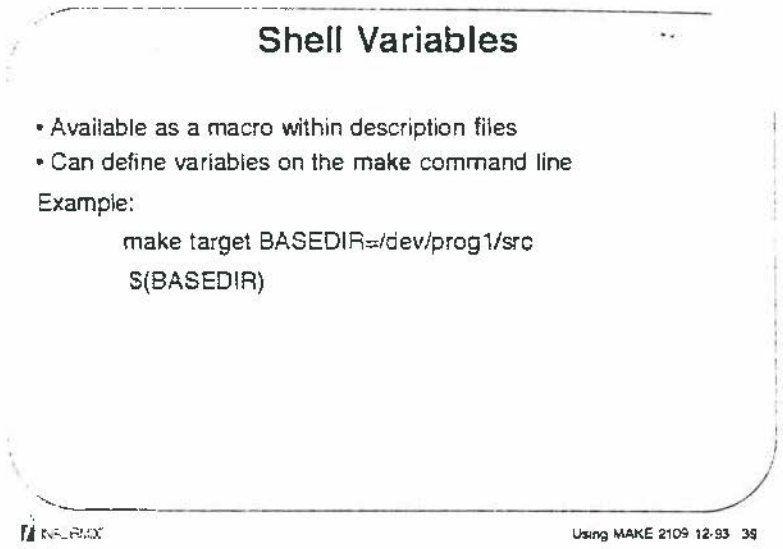
$(name)

Description file entries of the form **name = string** are macro definitions. Subsequent references to **$(*name*)** are interpreted as the value of the string. You can use brackets ({}), instead of parentheses in macro references.

A macro that is never explicitly defined, or has no string to the right of the equal sign, is assigned a null string. A macro reference in a dependency line must have been defined previously. Otherwise, the reference will yield a null string. You cannot define macros dynamically in command scripts; their definitions become fixed before command execution begins.

Internal Macros

The make utility has several internal macros that can simplify your description files. Refer to your UNIX system documentation for syntax and detailed descriptions of these macros.



Shell Variables

Shell variables that were part of the environment when you invoked **make** are available as macros within the description file. You refer to them using parentheses or braces whenever their names consist of more than one character.

You can also define variables on the **make** command line using the shell keyword command line facilities. The command above assigns the string **/dev/prog1/src** to the name BASEDIR. The string can be accessed within the description file as **$(BASEDIR)**.

Priority of Macro Assignments

When using macros in your description, consider the priority of macro assignments as follows:

- After the word **make** at the command line
- Description file definitions
- Shell environment variables or before the word **make** at the command line
- Internal macro definitions

Refer to your UNIX system documentation for more information.

INFORMIX

Makefile Sample

Define Suffixes

```
.SUFFIXES:

.SUFFIXES: .o .ec .c .4gl .4ge .4go .4gi .4ga .per .frm .msg .iem
```

Define 4GL, Forms and Executable Names

```
4GL = stub.4gl

FORM =

PROG_NAME = final_prog.4ge
```

Use Macro String Substitution to Define Other Objects

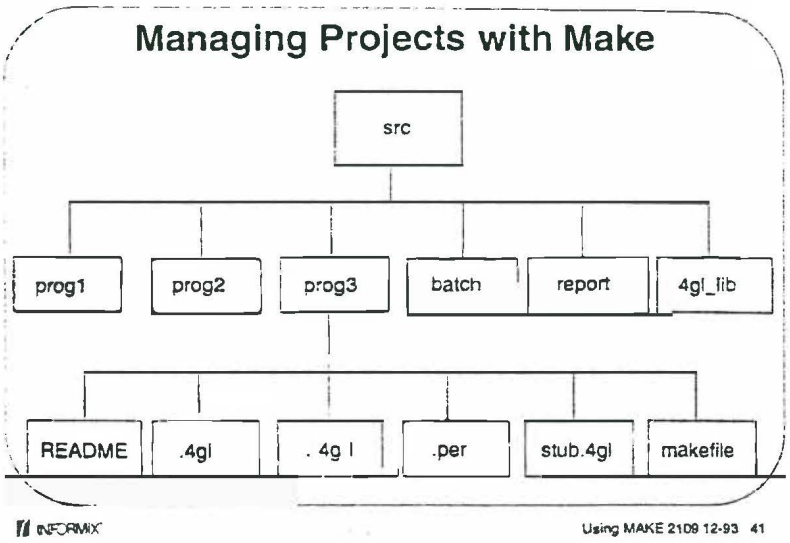
```
COBJS = $(4GL:.4gl=.o)
POBJS = $(4GL:.4gl=.4go)
CFORM = $(FORM:.per=.frm)
RDS_NAME= $(PROG_NAME:.4ge=.4gi)
```

Define the Dependencies

```
CCODE:  $(COBJS)
        c4gl -o $(PROG_NAME) $(COBJS)
        @- echo "Finished compiling ccode..."
PCODE:  $(POBJS)
        cat $(POBJS) > $(RDS_NAME)
        @- echo "Finished compiling pcode..."
FORM:   $(CFORM)
        @- echo "Finished compiling forms..."
```

Define the Compilation Rules

```
.4gl.4go:
        fglpc $*.4gl

.4gl.o:
        c4gl -c $*.4gl
        @- rm -f $*.c $*.ec

.per.frm:
        form4gl -s $*.per
```



Managing Projects with Make

The easiest and most efficient way to manage a project with many executables and many source files in many directories is to have a **makefile** in every development directory. The **makefile** in one development directory builds one executable program. Then, you can define a **makefile** at a higher level in the directory tree hierarchy that will build all of the programs.

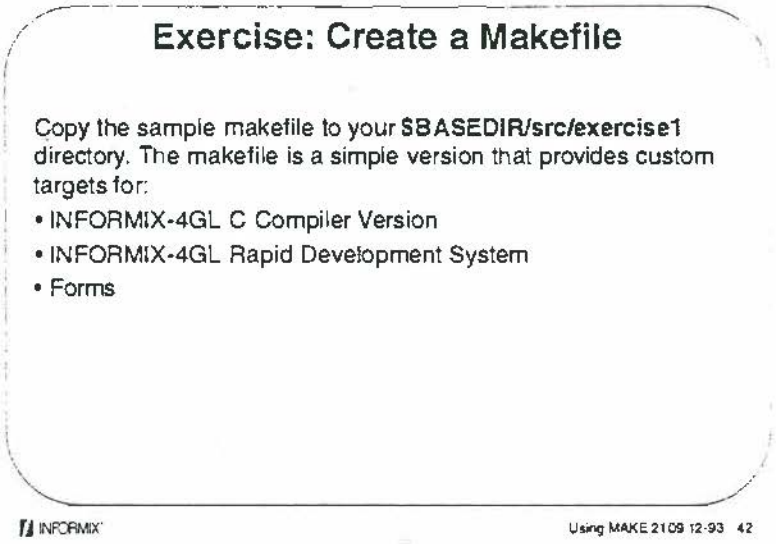
Use Include Files

The makefiles in the development directories should be simple and easy to maintain by the developer on a day-to-day basis. This is accomplished by using include files. Include files are separate files that contain the rules portion of the **makefile**.

Benefit

Include files provide ease in modification. If a rule change is required, the change is made to a single file and not 100 different developers' makefiles.

Sample makefiles that uses include files can be found in the Appendix section of this manual.


Exercise: Create a Makefile
Copy the sample makefile to your $BASEDIR/src/exercise1 directory. The makefile is a simple version that provides custom targets for:
• INFORMIX-4GL C Compiler Version
• INFORMIX-4GL Rapid Development System
• Forms
INFORMIX
Using MAKE 2109 12-93 42

Exercise: Create a Makefile

Copy the sample **makefile** to your **$BASEDIR/src/exercise1** directory. The **makefile** provides the following custom targets:

make CCODE	Compiles the INFORMIX-4GL C Compiler Version of the application.
make PCODE	Compiles the INFORMIX-4GL Rapid Development System of the application.
make FORM	Compiles the forms.

You will need to edit the **makefile** as you develop applications to define the name of your 4GL source files, forms and final executable.

The .ec suffix defined in the suffixes section is only required if you are using INFORMIX-ESQL/C.

Note: Complete makefiles can be found in the Appendix section of this manual.

Exercise: Using a 4GL Stub File

The code to implement this screen should consist of:

- A main module with a control (driver) function and a display function
- A second module with the SELECT function
- A stub module that calls the main module
- A form that displays customer number, first name, last name and company name.

Exercise: Using a 4GL Stub File

You should be in your **$BASEDIR/src/exercise1** directory to create your 4GL and form source files.

Step 1 Create a form that displays customer number, first name, last name, and company name from the **customer** table. Name the form **disp_win.per**.

Step 2 Create a module named **cust_ord.4gl** that has two functions: a control (or driver) function and a display function. Name the control function **cust_ctrl** and the display function **disp_cust**. **Cust_ctrl** should accept one parameter, the customer number.

Step 3 Create a second module named **svc_cust_ord.4gl** that performs the SELECT function. Name the select function **sel_cust_rec**.

Step 4 Create a stub module that uses **ARG_VAL()** to obtain the customer number from the command line and calls **cust_ctrl** with the customer number. Display a usage statement if the number of arguments is incorrect.

Step 5 Edit your **makefile** and define the names of your 4GL source modules and form.

Step 6 Run the CCODE and FORM targets to compile your application and forms.

Step 7 | Run the program with and without an argument.

Tip: | Use a PROMPT statement to keep the form on the screen.

Solutions

stub.4gl

main

```
DATABASE stores

MAIN

    DEFINE
        a_customer_num LIKE customer.customer_num

    IF ( NUM_ARGS ( ) != 1 ) THEN
        DISPLAY "USAGE: main.4ge <customer number>"
        DISPLAY ""
    ELSE
        LET a_customer_num = ARG_VAL(1)
        CALL cust_ctrl ( a_customer_num )
    END IF

END MAIN
```

cust_ord.4gl

cust_ctrl( )

```
DATABASE stores

FUNCTION cust_ctrl ( p_customer_num )

    DEFINE

        cust_rec    RECORD
        customer_num LIKE customer.customer_num,
        fname        LIKE customer.fname,
        lname        LIKE customer.lname,
        company      LIKE customer.company
        END RECORD,

        p_customer_num LIKE customer.customer_num,

        sql_stat INTEGER

    WHENEVER ERROR STOP

    OPEN WINDOW cust_win AT 2,3 WITH FORM "disp_win"
        ATTRIBUTE ( BORDER )

    CALL sel_cust_rec ( p_customer_num )
        RETURNING sql_stat, cust_rec.*

    IF ( sql_stat = NOTFOUND ) THEN
        ERROR "No customers found...."
        SLEEP 3

    ELSE
        CALL disp_cust ( cust_rec.* )

    END IF

    CLOSE WINDOW cust_win

END FUNCTION
```

disp_cust( )

```
FUNCTION disp_cust ( p_cust_rec )

    DEFINE

        p_cust_rec      RECORD
        customer_num    LIKE customer.customer_num,
        fname           LIKE customer.fname,
        lname           LIKE customer.lname,
        company         LIKE customer.company
        END RECORD,

        answer CHAR(1)

    DISPLAY BY NAME p_cust_rec.*

    PROMPT "Press any key to exit...." FOR CHAR answer

END FUNCTION
```

svc_cust_ord.4gl

sel_cust_rec( )

```
DATABASE stores

FUNCTION sel_cust_rec ( p_customer_num )

    DEFINE

        cust_rec     RECORD
        customer_num LIKE customer.customer_num,
        fname        LIKE customer.fname,
        lname        LIKE customer.lname,
        company      LIKE customer.company
        END RECORD,

        p_customer_num LIKE customer.customer_num,

        sql_stat INTEGER

    WHENEVER ERROR CONTINUE

    SELECT  customer.customer_num, customer.fname,
            customer.lname, customer.company
    INTO    cust_rec.*
    FROM    customer
    WHERE   customer.customer_num = p_customer_num

    WHENEVER ERROR STOP

    LET sql_stat = SQLCA.SQLCODE

    RETURN sql_stat, cust_rec.*

END FUNCTION
```

disp_win.per

```
DATABASE stores

SCREEN
{
 CUSTOMER NO: [f01      ] NAME:[f03             ][f02             ]
      COMPANY: [f04                  ]
}
END

TABLES
    customer
END

ATTRIBUTES

    f01 = customer.customer_num;
    f02 = customer.fname;
    f03 = customer.lname;
    f04 = customer.company;
END
```

INFORMIX

Chapter

4

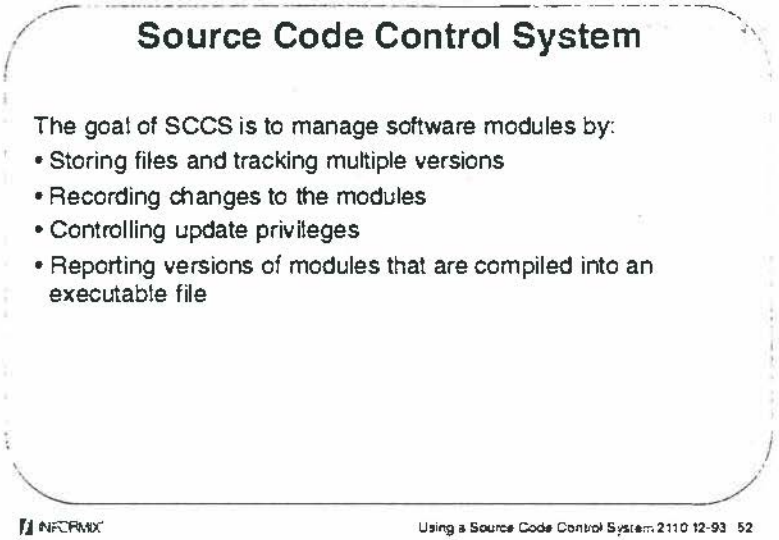
Using a Source Code Control System

Objectives

At the end of this module, you will be able to:

- Explain the benefits of using SCCS
- Describe the SCCS life cycle
- Execute basic SCCS commands
- Implement SCCS keywords in 4GL header files

© 1993 Informix Software, Inc.


Source Code Control System
The goal of SCCS is to manage software modules by:
• Storing files and tracking multiple versions
• Recording changes to the modules
• Controlling update privileges
• Reporting versions of modules that are compiled into an executable file
INFORMIX
Using a Source Code Control System 2110 12-93 52

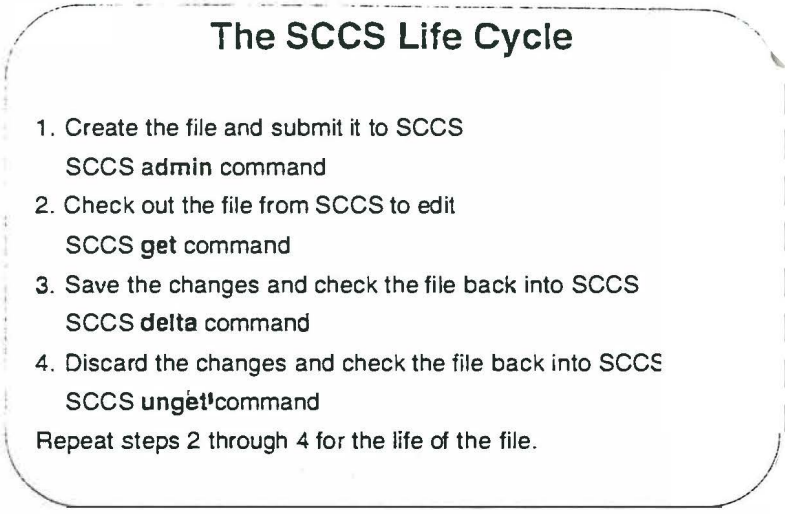
Source Code Control System

The overall goal of SCCS is to manage software modules. SCCS is a collection of programs for managing the development and maintenance of your programs. Its primary purpose is to automatically track changes between different versions of a program. SCCS records the following changes to the software module:

What	The lines updated, inserted and deleted in the module.
When	The date and time of the changes.
Who	The person that made the changes.
Why	The comments related to the change.

SCCS Scripts

SCCS is a UNIX utility. It is implemented on UNIX using shell commands. It is recommended to embed the SCCS commands in a shell script to simplify the commands. The Configuration Manager can be responsible for creating and managing user-friendly SCCS shell scripts.


The SCCS Life Cycle
1. Create the file and submit it to SCCS
SCCS admin command
2. Check out the file from SCCS to edit
SCCS get command
3. Save the changes and check the file back into SCCS
SCCS delta command
4. Discard the changes and check the file back into SCCS
SCCS unget command
Repeat steps 2 through 4 for the life of the file.
INFORMIX
Using a Source Code Control System 2110 12-93 53

The SCCS Life Cycle

A file is placed under control of SCCS by using the **admin** command and prefixing the filename with **s.***filename*. The source code file under SCCS is now called an ***s-file***. A file is submitted only once to SCCS, that is, the **admin** command is run only once for the life of the file.

Deltas (changes)

When a set of changes, known as a delta, is made to a file under SCCS, the changes are identified and stored along with the original source code and other administrative data. All of the information is kept in the **s-file**. SCCS automatically maintains a list of these changes and has the ability to recreate a particular version on request.

Conserves Disk Space

SCCS only tracks the changes between versions and not an entire copy of the version. Thus, it conserves disk space while maintaining multiple versions of code.

Placing Your Modules Under SCCS

```
admin -ifilename s.sccsname

admin -iq_order.4gl SSCCSDIR/s.q_order.4gl
```

- Run separately for each file
- Keep **s-files** in a separate directory
- Must begin with the characters **s.**
- Sccsname = filename, 12 characters maximum

Placing Your Modules Under SCCS

To place a software module under the SCCS system, you use the **admin** command. The basic format of the command is:

```
admin -ifilename s.sccsname
```

This command places the *filename* under SCCS, giving it the name **s.*sccsname***. The *sccsname* and *filename* should be the same. In some systems, the admin command removes the original copy of the file, while in others it leaves the original. If the original is left, you should remove it after ensuring that your **s-file** has been created.

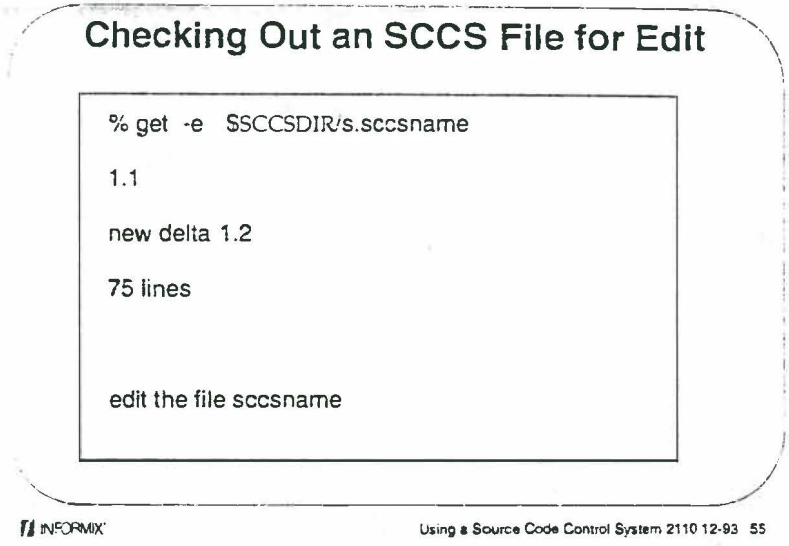
ignating an SCCS directory

You can designate a different SCCS directory by specifying the full path name for the **s.*sccsname*** file as follows:

```
admin -ifilename SSCCSDIR/s.sccsname
```

Note: It is recommended to set an environment variable, called **SCCSDIR**, to reference your SCCS directory path.

Warning! In a development environment from this point on, any access to your source files must be done through the SCCS system.



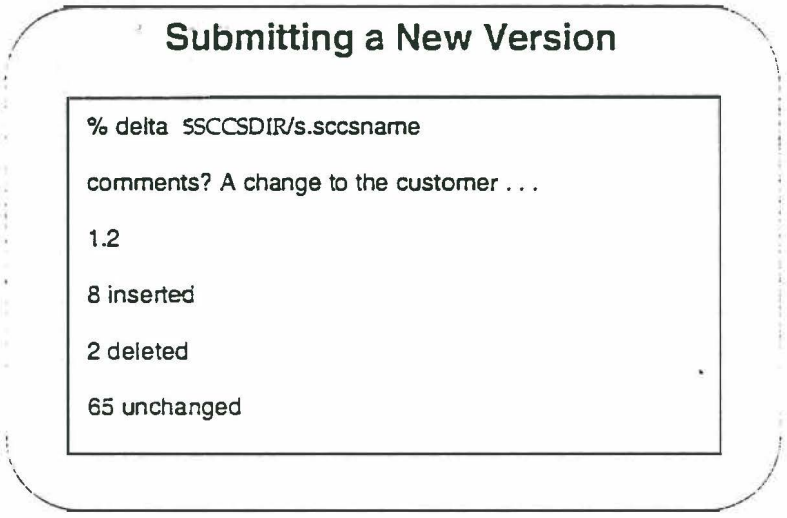
Checking Out an SCCS File for Edit

To retrieve an SCCS file to edit, use the **get** command with the **-e** flag. This gives you read and write permissions on the file, and prevents others from updating the file. The basic format of the command is:

```
get -e s.sccsname
```

This command will retrieve the file with the same name as the SCCS file, except without the leading s. characters. The file can be edited using a text editor. This command also creates a *p.sccsname* file, which contains information about who is editing the file, and locks the SCCS file against attempts by other developers to edit the file.

Note: The SCCS files cannot be edited directly. You should not vi the *s.sccsname* to make changes to the source code.



Submitting a New Version

To save the changes and check a file back into SCCS, use the delta command. The basic format of the command is:

```
delta s.sccsname
```

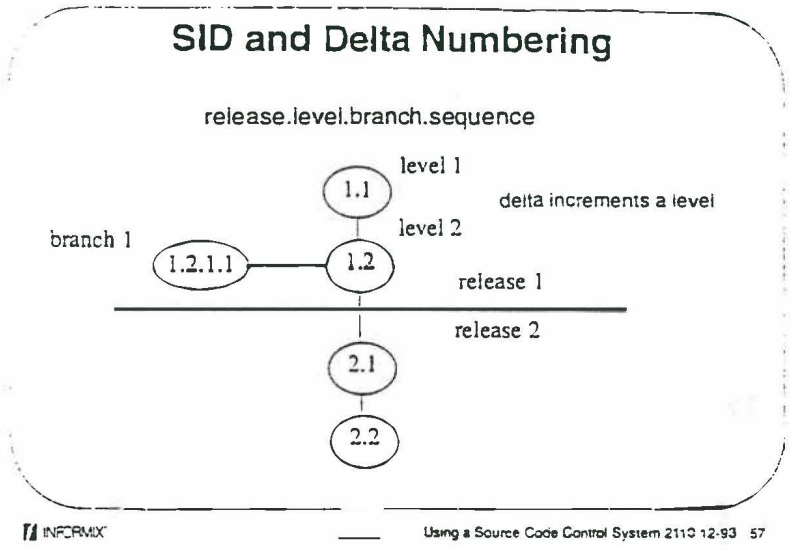
This command records the changes that you made to your file. It updates the current release level number and prompts for optional comments that will be stored with the SCCS file. Finally, it unlocks the SCCS file to allow others to check out the file.

The unget Command

If you do not want to save the changes you made to your file, but want to check the file back into SCCS, use the unget command. The basic format of the command is:

```
unget s.sccsname
```

This command ignores any changes that have been made to the file. It returns the SCCS file to the previous revision and unlocks the file.



SID and Delta Numbering

Each delta is assigned a name, called the SCCS *identification string* (SID). The SID is composed of at most four components: release, level, branch and sequence.

Deltas can be thought of as nodes of a tree in which the root node is the original version of the file. The root is normally named 1.1 and deltas (nodes) are named 1.2, 1.3, 1.4, etc. The components of these numbers are called release and level numbers, respectively. Normal naming of new deltas proceeds by incrementing the level number. This is done automatically by SCCS whenever a delta is made.

Release Numbers

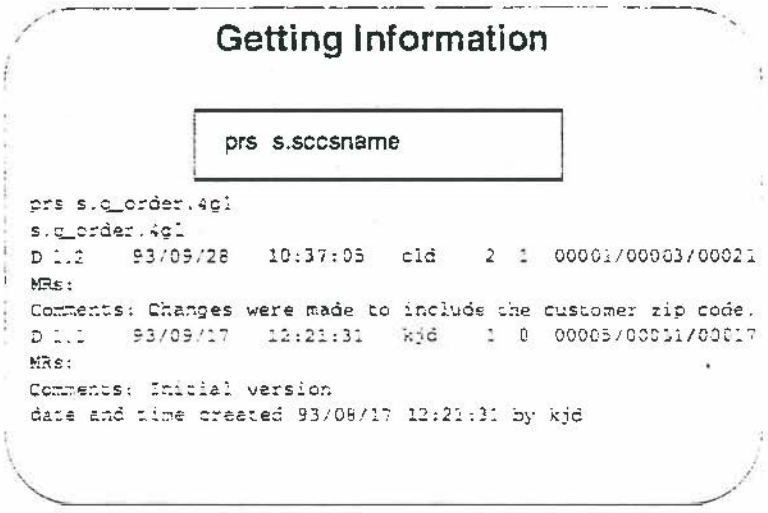
You may choose to change the release number to indicate a major build and create a new version of the program. The release number applies to all new deltas unless specifically changed again. This is done by using a -r option with the -e option in the get command as follows:

```
get -e -r2 s.g_order.4gl
```

Branches

There are situations that require branching an SCCS tree. That is, changes are planned to a given delta that will not be dependant on all previous deltas. Branch delta names always have all four SID components: the same release and level number as the trunk delta, plus a branch and sequence number.

Note: Although the branching capability is useful for certain specialized uses, the SCCS tree should be kept as simple as possible.

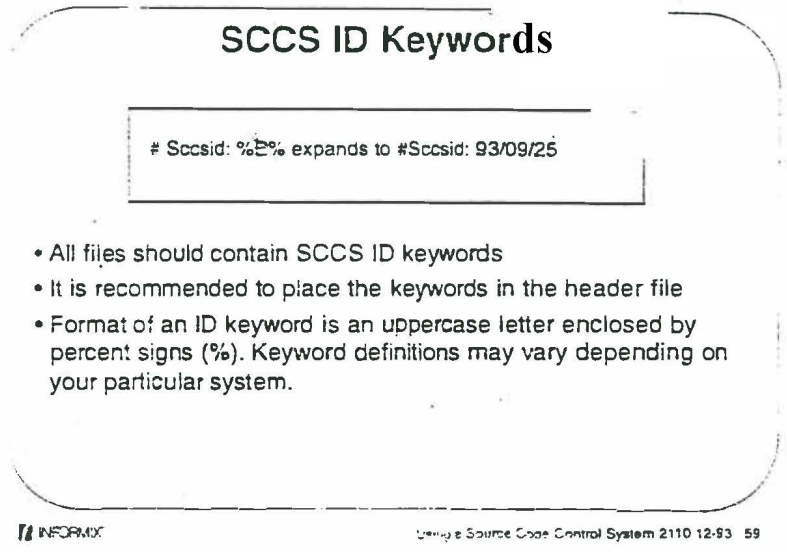

Getting Information
prs s.sccsname
prs s.c_order.4gl
s.c_order.4gl
D 1.2 93/09/28 10:37:05 cld 2 1 00001/00003/00021
MRs:
Comments: Changes were made to include the customer zip code.
D 1.1 93/09/17 12:21:31 kjd 1 0 00005/00001/00017
MRs:
Comments: Initial version
date and time created 93/08/17 12:21:31 by kjd
INFORMIX
Using a Source Code Control System 2110 12-93 58

Getting Information

The prs command provides information about your SCCS file. Running the command without any options displays a revision history. The information displayed in the example above is as follows:

Type of delta/change: R=delta was removed, D=delta exists
SCCS version number
Date the version was created
Time the version was created
Log-on id of the person who created the version
Sequence number of the version
Sequence number of the predecessor version
Number of new lines inserted in this version
Number of lines deleted in this version
Number of lines changed in this version
MRs = Modification Request numbers
Comments

Sequence numbers are serially assigned every time a change is made to the code, whether the change is a new release, level or branch.


SCCS ID Keywords
Sccsid: %E% expands to #Sccsid: 93/09/25
• All files should contain SCCS ID keywords
• It is recommended to place the keywords in the header file
• Format of an ID keyword is an uppercase letter enclosed by percent signs (%). Keyword definitions may vary depending on your particular system.
INFORMIX
Using a Source Code Control System 2110 12-93 59

SCCS ID Keywords

SCCS tracks files and executables they create by using keywords. All files submitted to SCCS must contain an SCCS ID keyword string. The string must contain at least one SCCS keyword.

The format of an ID keyword is an uppercase letter enclosed by percent signs (%). The letter in the middle has a special meaning and is expanded when the file is checked into SCCS.

Examples of keywords include:

%B% - SID Branch number

%E% - Date the newest delta was created, in the form of *yy/mm/dd*

%H% - Current date, in the form of *mm/dd/yy*

%I% - SID of the retrieved version, %B%%L%%R%%S%

%L% - SID Level number

%M% - Module name

%P% - SCCS file name (full path name)

%R% - SID Release number

%S% - SID Sequence number

%T% - Current time, in the form of *hh:mm:ss*

%W% - expands to the values of keywords %Z%%M%%T%%I%

%Z% - The string "@(#)" recognized by the **what** command

Keywords and INFORMIX-4GL

```
############################################################
#  Module              :  %M%
#  Description         :  holds all error and exception
#                         handling functions
#  Functions           :  fatal_err
#                         err_recover
#                         err_screen
#                         upd_err_log
#                         get_err_msg
#  SCCS Id No          :  %W%
#  Author              :  Ken Dowling (INFORMIX SOFTWARE)
#  Date                :  %H% %T%
#  Path                :  %P%
############################################################
```


Keywords and INFORMIX-4GL

All files submitted to SCCS should contain an SCCS ID keyword string. Keywords can put valuable information in your executable file. Most importantly, it can tell you the versions of the files that make up the executable. *This is very important for bug fixes and releases.* It is recommended to place the keywords in a comment in the header file as shown above.

%W%

The most commonly used string is the %W% keyword, which is expanded upon retrieval from SCCS into a character sequence that includes the %Z%%M%%T%%I% keywords.

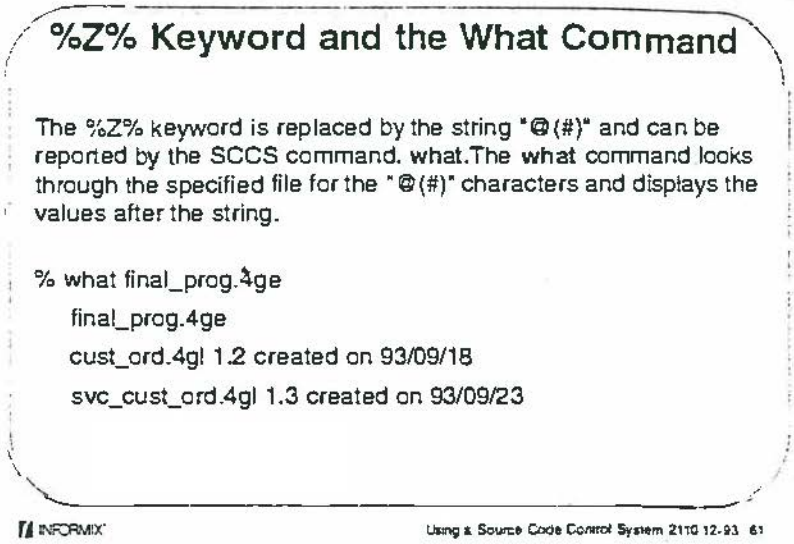
g_sccs_var Global Variable

A problem in INFORMIX-4GL development is that keywords placed in comments never reach the final executable. The compiler strips out all comments.

To implement keywords define a global variable to hold the SCCS keyword string **g_sccs_var CHAR (70)**.

Assign the keyword string to the global variable once for every module. It is recommended that you standardize the assignment of the global variable for all developers. You could assign the global variable as the last statement in the first function of every module:

```
LET g_sccs_var = "%W% created on %E%"
```

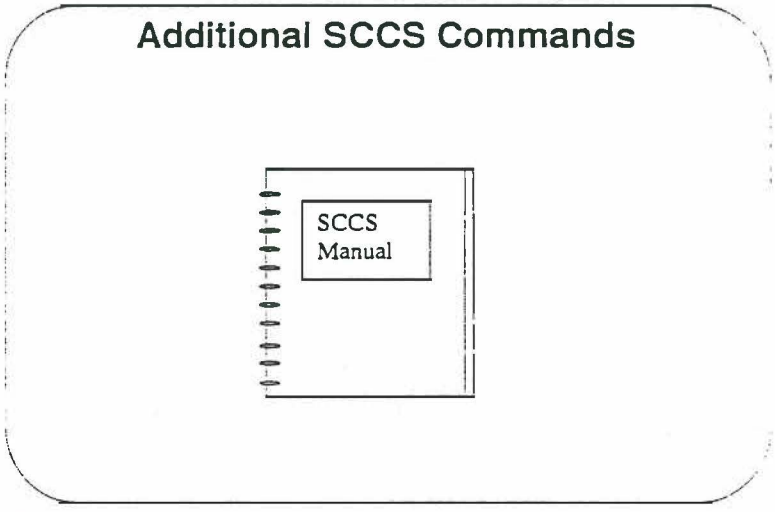

%Z% Keyword and the What Command
The %Z% keyword is replaced by the string "@(#)" and can be reported by the SCCS command. what.The what command looks through the specified file for the "@(#)" characters and displays the values after the string.
% what final_prog.4ge
final_prog.4ge
cust_ord.4gl 1.2 created on 93/09/18
svc_cust_ord.4gl 1.3 created on 93/09/23
INFORMIX
Using a Source Code Control System 2110 12-93 61

%Z% Keyword and the What Command

The %Z% keyword can put valuable information in your executable file. Most importantly it can tell you the versions of the files that make up the executable. *This is very important for bug fixes and releases.*

%Z% "@(#)"

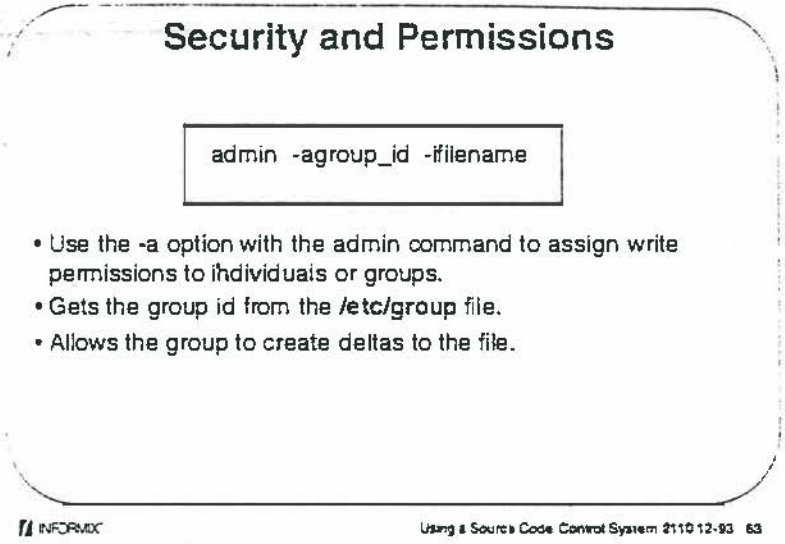
The %Z% keyword expands to a special character set which can be reported by the SCCS command what. The %Z% keyword is replaced by the string "@(#)". The what command looks through the specified file for the "@(#)" characters and displays the values after the string.



Additional SCCS Commands

Other SCCS commands are explained in the SCCS reference documentation produced by the UNIX operating system vendors. The following commands are explained briefly:

help	Explains error messages generated from a command.
sact	Prints information about files that are currently out for edit.
rmdel	Removes a delta of an SCCS file that was created in error.
sccsdiff	Shows differences between any two versions of an SCCS file.
cdc	Changes the comment information associated with a delta.


Security and Permissions
admin -agroup_id -ifilename
• Use the -a option with the admin command to assign write permissions to individuals or groups.
• Gets the group id from the /etc/group file.
• Allows the group to create deltas to the file.
INFORMIX
Using a Source Code Control System 2110 12-93 63

Security and Permissions

SCCS relies on the capability of the UNIX system for most of the protection mechanisms required to prevent unauthorized changes to the SCCS files (i.e. changes by non-SCCS commands).

Files created by the **admin** command are given read-only access permission. This mode should remain unchanged to prevent modification of SCCS files by non-SCCS commands.

The -a option to the **admin** command can be used to grant write permission to one or more individuals or to a group defined in the /etc/group file.

Exercise: Using SCCS

```
#############################################################
#  Module              :  %M%
#  Description         :  Main control function and screen
#                         interaction for displaying
#                         customer order information.
#
#  Functions           :  cust_ctrl
#                         disp_cust
#  SCCS Id No          :  %W%
#  Author              :  Ken Dowling (INFORMIX SOFTWARE)
#
#############################################################
```


Exercise: Using SCCS

This exercise is intended to introduce you to SCCS. Perform the following steps listed below to implement your 4GL source modules under SCCS.

It is recommended that your Configuration Manager (or appropriate individual) create a user friendly shell script and embed the SCCS commands and administrative tasks within the script. The shell script should remove the complexity and day-to-day management tasks of SCCS from the programmer.

Step 1 Set an environment variable, called SCCSDIR, to reference your SCCS directory path: **SBASEDIR/sccs/exercise1.**

Step 2 Use the **admin** command to place your 4GL source modules from the previous exercise into the SCCS directory. You will need to specify the full path name of your SCCS directory using SCCSDIR.

Step 3 If you are sure the **admin** command was successful, change the mode of your software modules to read only:

```
chmod 444 *.4gl
```

Note: In some systems, the admin command removes the original copy of the file, while in others it leaves the original. If the original is left, you can remove it after ensuring that your s-file has been created, or you

can change the mode to read only.

SCCS will not allow a writable copy of your software module to exist outside of the SCCS structure after the file has been placed under SCCS.

Step 4 Use the **get** command to retrieve your 4GL source files for edit. You will need to specify the full path name of the SCCS directory.

Step 5 To implement keywords to use the **what** command, create a globals file and define a global variable to hold the SCCS keyword string: **g_sccs_var** CHAR (70). Name the globals file **globals.4gl**.

Step 6 Use the **admin** command to place your **globals.4gl** file under SCCS. Change the mode of the file to read only:

```
chmod 444 globals.4gl
```

Step 7 Edit your 4GL source modules:

- Include the header file shown on the previous page. Modify the header file to reflect the correct information.
- Include your global file declaration.
- Assign the keyword string to the global variable once for every module. Assign the global variable as the last statement in the first function of both modules:

```
LET g_sccs_var = "%W% created on %E%"
```

Step 8 Use the **delta** command to check in your 4GL source files and save the changes.

Step 9 Use the **get** command without the -e (edit) option to get all of your software modules. A read only copy of the software module is required in order to recompile the application.

Step 10 Edit your **makefile** and define your globals file name on your 4GL source line.

Step 11 Recompile the application using the CCODE target.

Step 12 Run the **what** command on the executable. Notice that **what** reports on your own software modules first.

Solutions

Step 1

```
% setenv SCCSDIR  $BASEDIR/sccs/exercise1
```

Step 2

Repeat for each 4GL module:

```
admin -i<filename> $SCCSDIR/s.<filename>
```

Step 3

```
chmod 444 *.4gl
```

Step 4

Repeat for each 4GL module:

```
get -e $SCCSDIR/s.<filename>
```

Step 5

globals.4gl

```
GLOBALS
    DEFINE
        g_sccs_var CHAR (70)
END GLOBALS
```

Step 6

```
admin -iglobals.4gl $SCCSDIR/s.globals.4gl
```

Step 7

Include the header file in each module. Add the globals declaration following the DATABASE statement:

```
GLOBALS 'globals.4gl'
```

Assign the keyword string to the global variable as the last statement in the first function of each module:

```
LET g_sccs_var = '%W% created on %E%'
```

Step 8

Repeat for each 4GL module:

```
delta $SCCSDIR/s.<filename>
```

Step 9

Repeat for each 4GL module

```
get $SCCSDIR/s.<filename>
```

Step 10

4GL = globals.4gl stub.4gl cust_ord.4gl svc_cust_ord.4gl

Step 11

```
make CCODE
```

Step 12

```
% what final_prog.4ge
```

Output:

```
final_prog.4ge:

cust_ord.4gl 1.2 created on 93/09/18
svc_ cust_ord.4gl 1.3 created on 93/09/23
```

Chapter

Using the Interactive Debugger

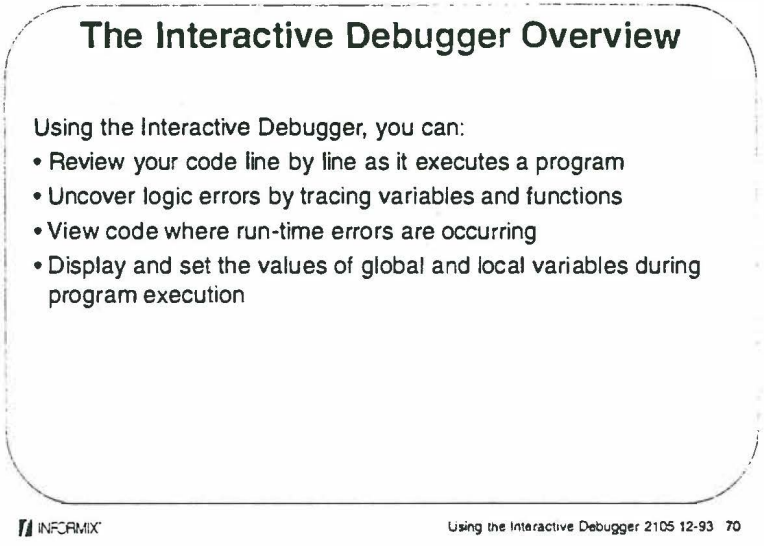
5

Objectives

At the end of this module, you will be able to:

- Explain the features and benefits of the Interactive Debugger
- Execute Interactive Debugger commands
- Use the Interactive Debugger to debug a 4GL program

© 1993 Informix Software, Inc.


The Interactive Debugger Overview
Using the Interactive Debugger, you can:
• Review your code line by line as it executes a program
• Uncover logic errors by tracing variables and functions
• View code where run-time errors are occurring
• Display and set the values of global and local variables during program execution
INFORMIX
Using the Interactive Debugger 2105 12-93 70

The Interactive Debugger Overview

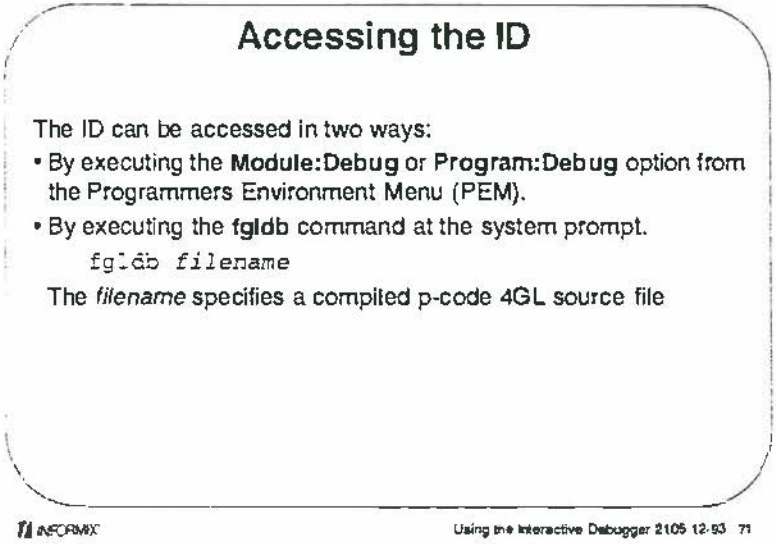
The Interactive Debugger (ID) has features that make it an extremely valuable tool for debugging applications. You can use the debugger to correct program logic errors and fatal errors. The ID allows you to:

- review source code as it executes, stepping through the program line by line.
- interrupt a program and resume execution of a program.
- specify break points to suspend program execution at a specific line of code or function.
- evaluate the values of program variables, check how a variable is defined, and set the value of a variable.

Not Just for Debugging

The ID can be used for educational purposes too. Running source code through the ID is an excellent way to familiarize yourself with the code. For new 4GL programmers, watching how code actually executes using the ID is a valuable learning experience.

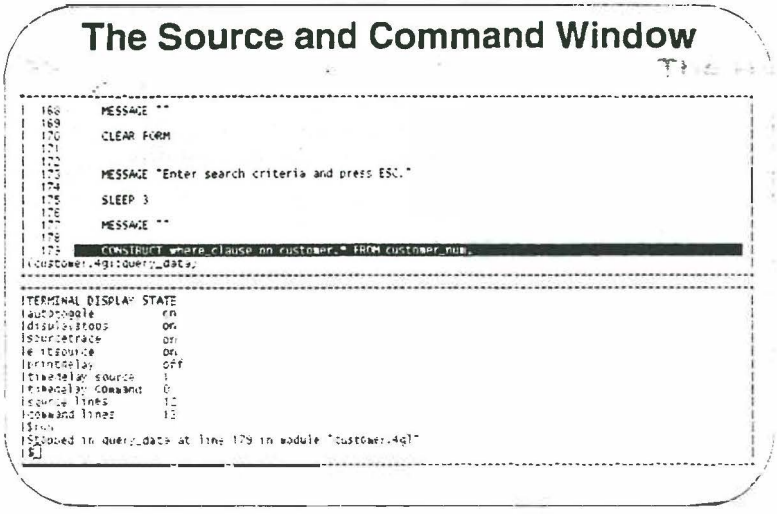
This module will introduce several debugging techniques using the ID. For additional information about the ID, refer to the INFORMIX-4GL *Interactive Debugger Manual.*


Accessing the ID
The ID can be accessed in two ways:
• By executing the Module:Debug or Program:Debug option from the Programmers Environment Menu (PEM).
• By executing the fgldb command at the system prompt.
fgldb filename
The filename specifies a compiled p-code 4GL source file
INFORMIX
Using the Interactive Debugger 2105 12-93 71

Accessing the ID

There are two ways to access the ID. You can execute the appropriate option from the PEM or execute the **fgldb** command at the system prompt. When a program is first loaded into the ID, the module containing the program's MAIN function is displayed to the **Source** window. The cursor is in the **Command** window and waiting for input.

Tip: You can execute operating system commands from within the ID. Enter the command at the prompt preceded by a exclamation point (!).



The Source and Command Window

The ID has two screens, the *application screen* and *debugger screen*. The application screen display is exactly the same as if you were executing the application program as a stand alone program.

To enter the debugger screen, you must suspend the execution of the program. The **interrupt** key will suspend the execution of the program, exit the application screen and display the debugger screen. The **continue** command resumes program execution.

The debugger screen consists of two windows:

- The **Source** window
  - The **Source** window is used to display the currently executing program source code segment. A highlighted bar indicates the next source code line that will be executed when the program resumes. You can scroll through the **Source** window using the UP and DOWN **arrow** keys.
- The **Command** window
  - The **Command** window is used to enter debugger commands. The output of the commands are displayed in this window. The cursor appears at the **$** prompt in the window, and the debugger waits for input from the user. You can scroll through the **Command** window.

Starting the Debugger

You start the debugging session by entering the run command at the Command window prompt. If the program requires arguments, list the arguments after the run keyword. The run command starts the program execution. You end the debugging session by entering the exit command at the Command window prompt.

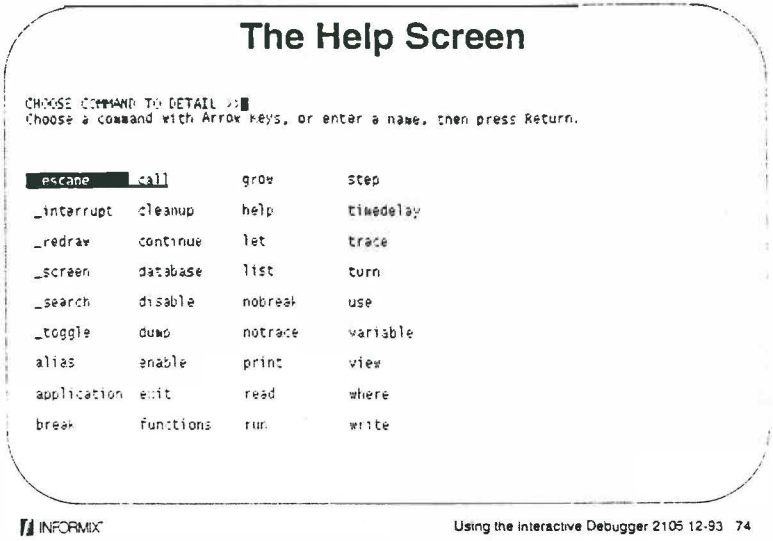
The Grow Command

Grow is a screen management command that specifies the relative size of the Command and **Source** windows. You can specify the **grow** command as follows:

grow [SOURCE | COMMAND] n_lines

Examples:

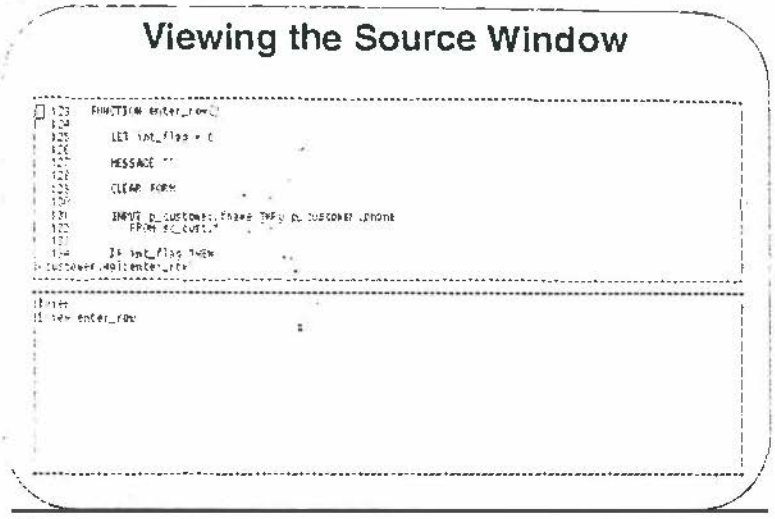
```
grow source 3
grow command 4
grow 5
```



The Help Screen

The **help** command displays the help screen and a synopsis of every debugger command. To invoke the help screen, you can:

- Enter the **help** command at the **Command** window prompt. If you do not specify a debugger command after the **help** keyword, the help screen displays a list of debugger commands.
- Enter the **help** command followed by a debugger command at the **Command** window prompt. The help screen displays an explanation of the debugger command.



Viewing the Source Window

The view command is used to move the cursor from the Command window to the Source window. The INTERRUPT key is used to return the cursor to the Command window. You can pass a function or module name to the view command to view a specific function or module.

Scrolling the Source Window

You can scroll through the Source window by using the UP and DOWN arrows keys or by the following control keys:

CTRL-K	move cursor up one line
CTRL-J	move cursor down one line
CTRL-B	move cursor up one window
CTRL-F	move cursor down one window
CTRL-U	move cursor up one-half window
CTRL-D	move cursor down one-half window

Character Patterns and Wildcards

You can search the Source window for a particular pattern of characters with the following keys:

/	searches forward from the current postion
?	searches backward from the current position
*	matches any subset of a string
?	matches any single character

Sourcetrace

Sourcetrace is a terminal display parameter that is very useful when familiarizing yourself with a 4GL program. If **sourcetrace** is enabled, the debugger highlights each line of code as it executes.

To enable **sourcetrace**, enter the command at the **Command** window prompt:

```
turn on sourcetrace
```


Sourcetrace

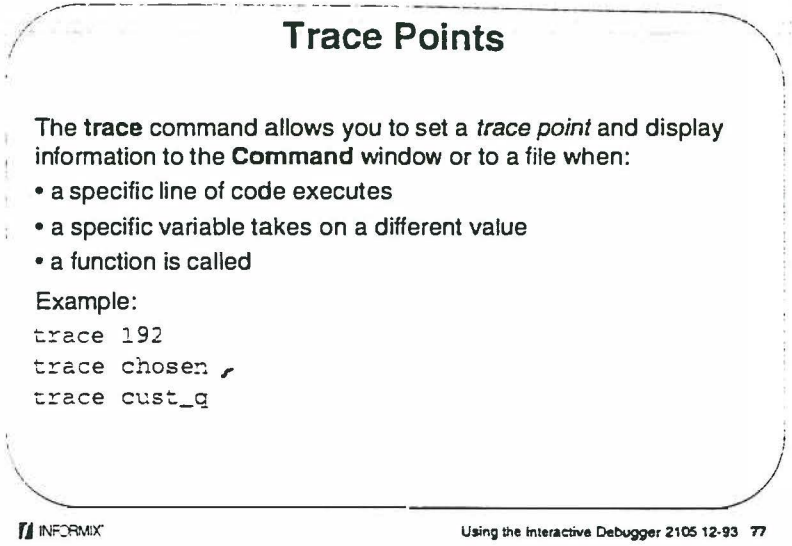
The **sourcetrace** command is very useful when familiarizing yourself with a 4GL program. **Sourcetrace** increases the time required for the debugging session. The **Source** and **Command** windows remain on the screen, and each line of source code is highlighted as the code executes. The debugger switches to the application screen only when user input is required.

The **sourcetrace** command is not enabled by default. To enable **sourcetrace**, enter the command:

```
$ turn on sourcetrace
```

To disable sourcetrace, enter the command:

```
$ turn off sourcetrace
```


Trace Points
The trace command allows you to set a trace point and display information to the Command window or to a file when:
• a specific line of code executes
• a specific variable takes on a different value
• a function is called
Example:
trace 192
trace chosen
trace cust_q
INFORMIX
Using the Interactive Debugger 2105 12-93 77

Trace Points

You can set specific *trace points* in your 4GL program. When the debugger encounters the specified trace points, it displays information to the Command window or to a file and continues program execution. The information includes the line number, function name and module name where the trace point was encountered.

You can specify trace points at the Command window prompt as follows:

trace *lineno*	trace 192
trace *variable*	trace chosen (must be a GLOBAL variable)
trace *function*	trace cust_q

Trace Points and Local Variables

To set a trace point for a local variable, you must specify the function name or display the function in the Source window. There are two ways to qualify the variable:

```
$ trace (function_name) variable
$ trace function.function_name.variable
```

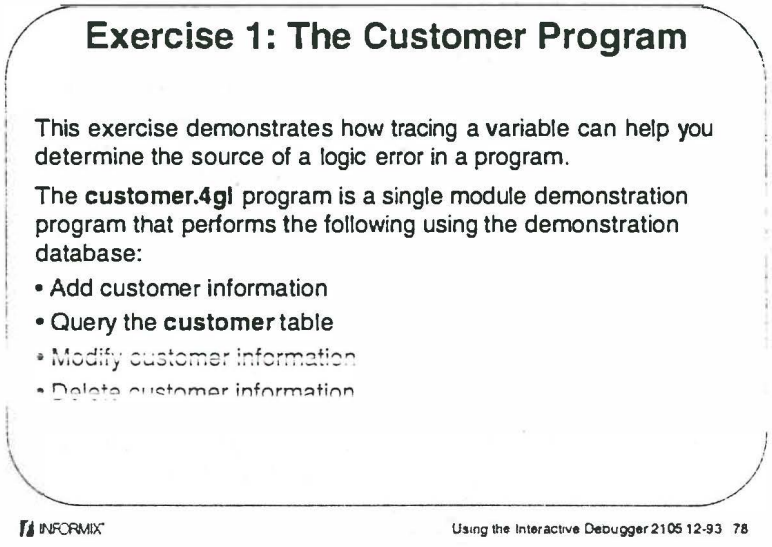
The Output File

You can send the information from the trace point to a file by using the >> symbol and specifying a file name as follows:

```
$ trace functions >> filename.out
```

Notrace

Use the **notrace** command to remove a trace point.


Exercise 1: The Customer Program
This exercise demonstrates how tracing a variable can help you determine the source of a logic error in a program.
The customer.4gl program is a single module demonstration program that performs the following using the demonstration database:
• Add customer information
• Query the customer table
• Modify customer information
• Delete customer information
INFORMIX
Using the Interactive Debugger 2105 12-93 78

Exercise 1: The Customer Program

In order to use the Interactive Debugger, the program must be compiled using INFORMIX-4GL Rapid Development System. Perform the following steps below:

Step 1

Change directories to **$INFORMIXDIR/demo**. Copy the **customer.4gl** module and **customer.per** form file to your **demo1** directory. Edit the DATABASE statement in **customer.4gl** and **customer.per** to reflect the name of your database.

Step 2

Copy your **makefile** from your source directory to the **demo1** directory. Edit the **makefile** and define the 4GL source and form file names: **customer.4gl customer.per.**

Step 3

Compile the application using the PCODE target and compile the form using the FORM target.

make PCODE

make FORM

Step 4

Start the debugging session for the customer program by invoking the ID at the command line.

```
fgldb customer
```

Step 5 Run the program and attempt the following:

- Choose the **Customer:Delete** option. Notice that you get a message that you must query a customer record before you can execute the Delete option.
- Query for a customer. Press y to select customer **102**.
- Delete customer **102**. Press y to verify the delete.
- Choose the **Delete** option again. Press y to verify the delete. Ideally, a user would not be able to perform a delete when a row is not displayed on the screen. Being able to do so is confusing.

Step 6 Exit the customer program.

```
Ctrl-C
```

Step 7 View the **show_menu** function that calls the **Customer:Delete** menu option to see what code is being executed when the user chooses to delete a row.

```
$ view show_menu
```

Step 8 Browse through the code until you see the code for the **Delete** option. Read the code. Notice that the user is prompted to verify the delete if the variable **chosen** is set to **TRUE**. Otherwise, the user is not allowed to perform the delete.

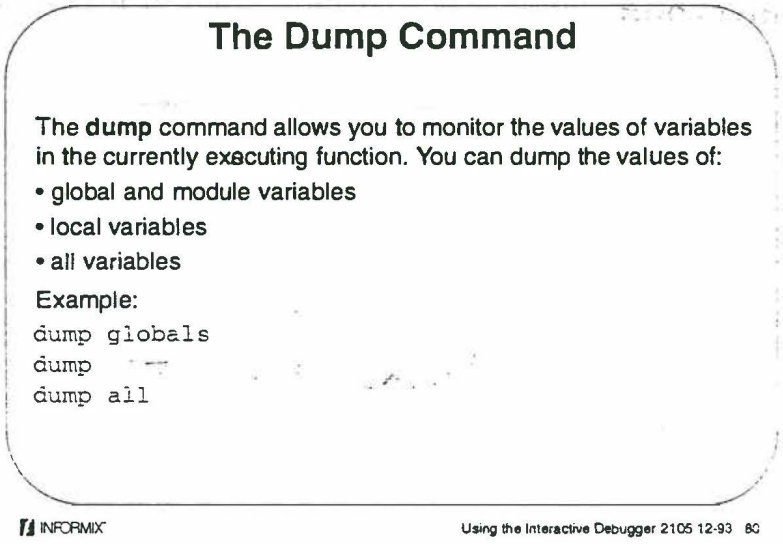
Step 9 Trace the variable chosen to see where in the program its value is begin changed from its initial value, FALSE (0) to TRUE (1).

```
$ trace chosen
```

Step 10 Repeat the instructions in **Step 5**, except select and delete customer **103**. After you have run the steps, exit the program.

In the output text region, the debugger displays the value if chosen was changed to TRUE in the **query_data** function. This is what we would expect; the user queries for a row, and chosen is set to TRUE because a row is displayed on the screen for the user to delete.

Step 11 View the **Delete** option in the **show_menu** function again. Determine what the logic error is in the **Delete** option.


The Dump Command
The dump command allows you to monitor the values of variables in the currently executing function. You can dump the values of:
• global and module variables
• local variables
• all variables
Example:
dump globals
dump
dump all
INFORMIX
Using the Interactive Debugger 2105 12-93 80

The Dump Command

You can display the value of variables in the currently executing function by using the **dump** command. It displays the values to the **Command** window or to a file.

You can specify to dump the values of global and module variables, local variables, or all variables at the **Command** window prompt as follows:

dump	displays the values of all local variables.
dump globals	displays the values of all global variables and the values of **int_flag**, **quit_flag**, status and SQLCA.
dump all	displays the values of all variables.

The Output File

You can send the information from the **dump** command to a file by using the **>>** symbol and specifying a file name as follows:

```
$ dump globals >> filename.out
```

Running the Dump Command

To use the **dump** command, your program must be running. This can be done two ways:

1. Interrupt the program, enter the **dump** command and then resume program execution.
2. Combine the **dump** command with the **trace** command.

Example:

```
$ trace functions {dump globals}
```

Break Points

The real power of the ID is realized when break points are set in an application. Break points allow you to suspend program execution at a designated point in the program, and evaluate the status of the source code to that point.

You can set a break point to suspend execution automatically when:

- a specific line of code executes
- a specific variable takes on a different value
- a function is called
- a particular expression evaluates to TRUE

INFORMIX Using the Interactive Debugger 2105 12-93 81

Break Points

The break command is a powerful command that allows you to set a break point and suspend program execution to perform diagnostics. When the debugger encounters a break point, it suspends program execution automatically and highlights the next statement to be executed in the Source window.

The cursor is placed in the Command window waiting for input. Information is displayed to the Command window. The information includes the line number, function name and module name where the break point was encountered. The program will not resume execution until specified to do so with the continue command.

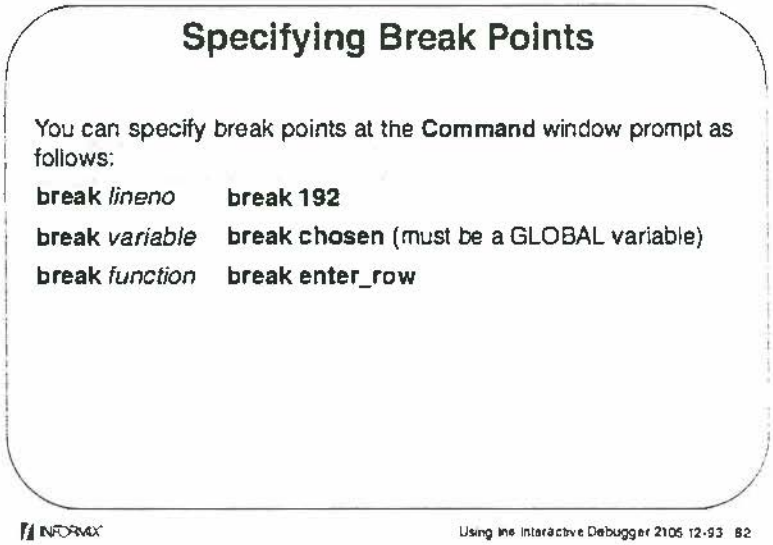
Nobreak

You can specify the nobreak command to remove the break point as follows:

```
$ nobreak reference_no
```

where *reference_no* is the reference number of the break point indicated by a number enclosed in parenthesis (*n*).

Tip: You should disable or turn off the trace point prior to running a program with a break point. This will allow you to observe the break points clearly.


Specifying Break Points
You can specify break points at the Command window prompt as follows:
break lineno
break 192
break variable
break chosen (must be a GLOBAL variable)
break function
break enter_row
INFORMIX
Using the Interactive Debugger 2105 12-93 82

Specifying Break Points

To set a break point for a local variable, you must specify the function name or display the function in the Source window. There are two ways to qualify the variable:

1. break (*function_name*) *variable*
2. break function.*function_name.variable*

Specifying a Condition

You can qualify a break point for a line number, variable or function by specifying a condition for program execution to be suspended.

Example:

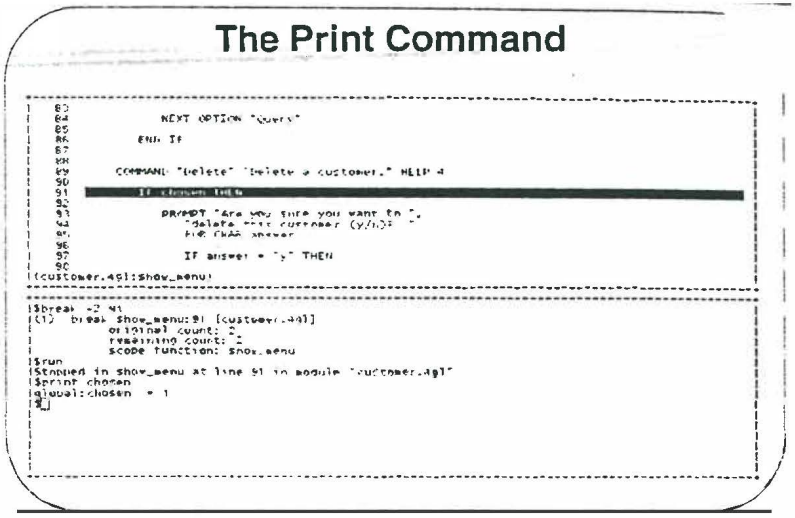
```
$ break if chosen = TRUE
$ break delete_row if p_customer.customer_num > 102
```

Specifying a Count

You can qualify a break point by specifying the number of times the condition should occur before suspending program execution. You can specify a count by indicating a number preceded by a minus sign -. The count is decremented by the debugger each time the condition is encountered.

Example:

```
$ break -3 if chosen = TRUE
$ break -2 delete_row if p_customer.customer_num > 102
```



The Print Command

The print command allows you to evaluate the values of individual variables in the currently executing function. You can specify to print the values of a variable to the **Command** window as follows:

print *variable*	displays the value of the variable.
print *program_record*	displays the values of a program record.
print *program_array*	displays the value of a program array.

Printing the Variable

To print the value of a variable, it must be in the active function, or you must specify the function name or display the function in the Source window. You can qualify the variable as follows:

```
$ print function.function_name.variable
```

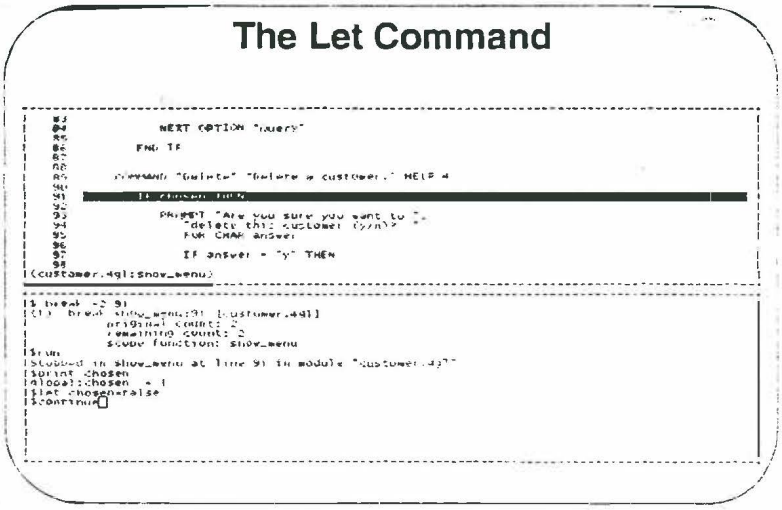
Continuing After a Break Point

After you reach a break point, you can continue program execution at the point of interruption by executing the **continue** command, or you can *step* through the application.

The Output File

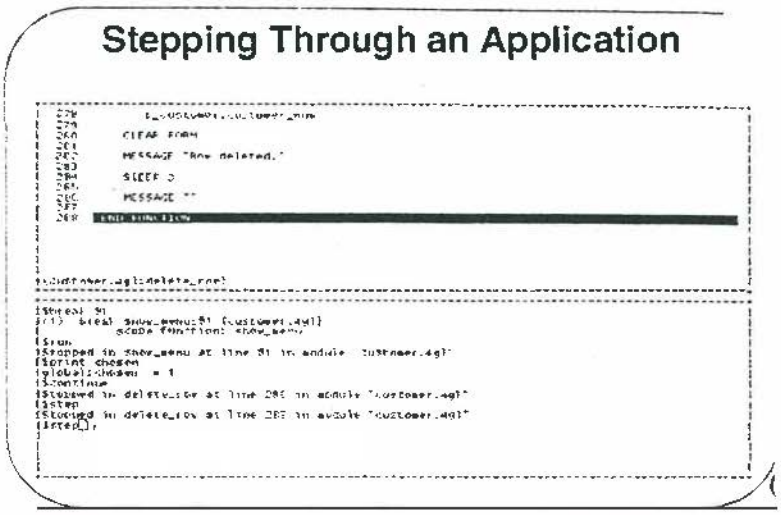
You can send the information from the print command to a file by using the >> symbol and specifying a file name as follows:

```
$ print chosen >> filename.out
```



The Let Command

The **let** command is a powerful debugging statement that allows you to interactively set the value of a variable while the program is suspended and then resume program execution with the new value. You can assign the **let** statement at the Command window prompt.



Stepping Through an Application

The **step** command allows you to step through the execution of a program. You can specify the **step** command at the **Command** window prompt as follows:

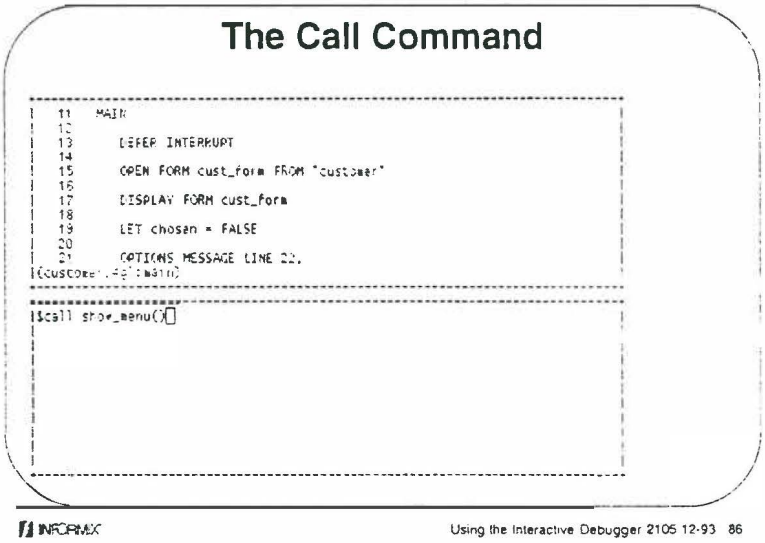
```
$ step n
```

where the letter *n* specifies the number of statements you want to execute. If a number is not specified, the debugger will step through the application one statement at a time.

The Into Option

When the next statement is a function call, you can use the **into** option to tell the debugger to move through the individual statements of the function.

```
$ step into
```



The Call Command

The **call** command allows you to execute a function interactively. If the function requires arguments, list the arguments between parentheses. You can specify the **call** command at the **Command** window prompt as follows:

```
$ call function ( [arg1, arg2...])
```

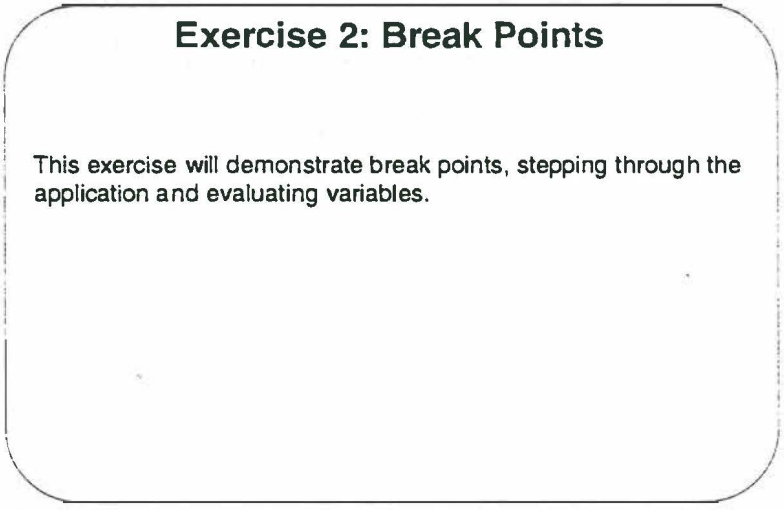
The parentheses are required even if the function requires no arguments.

Keyboard Shortcuts

Debugger Commands

break	b
call	ca
continue	co
interrupt	co i
quit	co q
dump	du
dump all	du a
dump globals	du g
exit	ex
functions	f
let	le
list	li
break	li b
display	li d
trace	li t
nobreak	nob
notrace	not
print	p
run	ru
step	s
into	s i
trace	tr
functions	tr functions
view	vi

control C
control T
control X


Exercise 2: Break Points
This exercise will demonstrate break points, stepping through the application and evaluating variables.
INFORMIX
Using the Interactive Debugger 2105 12-93 88

Exercise 2: Break Points

If the ID is not loaded, start the debugging session for the **customer** program by invoking the ID at the command line and perform the steps described below.

Step 1 — Insert a break point at line 92.

```
$ break 92
```

Step 2 — Run the program and query for customer **107**. Enter **y** to select customer **107**.

Step 3 — Press the **Delete** option. This should return you to the main window of the ID.

Step 4 — Evaluate the **chosen** variable. What is it set to?

```
$ print chosen
```

Step 5 — Step through the application and choose to delete the row when prompted.

Step 6 — After execution of the delete, interrupt the program and return to the main window of the ID.

```
Ctrl-c
```

Step 7

Evaluate the **chosen** variable. If **chosen** were set to FALSE at this point the program would run correctly. To verify this, set **chosen** to FALSE and continue the execution of the program. Notice now the delete is not allowed in the program.

```
$ let chosen = false
$ continue
```

Evaluating Construct Statements

Another very useful feature of the ID is the ability to evaluate query strings resulting from a CONSTUCT statement. Building the CONSTRUCT string often leads to syntax errors if not constructed properly.

Step 1

Insert a break point at line 191.

Step 2

Run the program and attempt the following:

- Choose the **Customer: Query** option.
- Enter some selection criteria and press the **Escape** key. This should return you to the main window of the ID.
- Evaluate the **sql_stmt** variable string.

Chapter

6

Creating Advanced Screen Arrays

Objectives

At the end of this module, you will be able to:

- Create a FORMONLY form specification file
- Define a screen array form
- Create a screen form that displays multiple arrays and allows users to move between the arrays and scroll through a set of data
- Control the cursor movement within multiple arrays
- Highlight the current row in an array

© 1993 Informix Software, Inc.

FORMONLY Fields

```
DATABASE FORMONLY

SCREEN
{
 CUSTOMER NO: [F01    ]          NAME: [F03              ][F02              ]
     COMPANY: [F04              ]
---------------------- ORDER INFORMATION ----------------------------
  Order Num   Order Dt   PO Num        Ship Dt     Ship Chg   Paid Dt
  [F05    ] [F06     ] [F07        ] [F08     ] [F09    ] [F10     ]
  [F05    ] [F06     ] [F07        ] [F08     ] [F09    ] [F10     ]
  [F05    ] [F06     ] [F07        ] [F08     ] [F09    ] [F10     ]
  [F05    ] [F06     ] [F07        ] [F08     ] [F09    ] [F10     ]
  [F05    ] [F06     ] [F07        ] [F08     ] [F09    ] [F10     ]
  [F05    ] [F06     ] [F07        ] [F08     ] [F09    ] [F10     ]
}
END

ATTRIBUTES

    F01 = FORMONLY.customer_num;
    F02 = FORMONLY.lname;
    F04 = FORMONLY.company;
    F05 = FORMONLY.order_num;
    F06 = FORMONLY.order_date TYPE DATE, FORMAT = "MM/DD/YY";
    F07 = FORMONLY.po_num;
    F08 = FORMONLY.ship_date TYPE DATE, FORMAT = "MM/DD/YY";
    F09 = FORMONLY.ship_charge TYPE MONEY, FORMAT = "$###.##";
    F10 = FORMONLY.paid_date TYPE DATE, FORMAT = "MM/DD/YY";

END

INSTRUCTIONS

    SCREEN RECORD sa_ord_rec[6] ( order_num, order_date, po_num, ship_date,
                                  ship_charge, paid_date );

END
```


FORMONLY Fields

FORMONLY fields are not associated with columns of any database table. They are used to enter or display values of program variables.

Database Section

If the DATABASE section specifies FORMONLY:

- No TABLES section is required.
- Field descriptions that you can specify in the ATTRIBUTES section must be FORMONLY.

Attributes Section

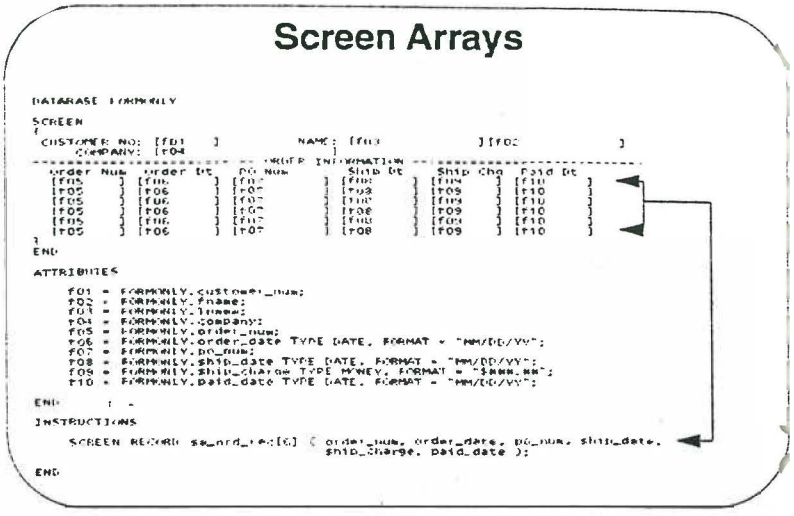
You must specify a data type only if:

- You use the INCLUDE or DEFAULT attribute for the field.
- You use a FORMAT attribute for a NUMERIC or DATE data type.

Form4gl assumes the field is a CHAR data type whose length is the width of the field. INFORMIX-4GL performs the necessary data conversion for the corresponding program variable.

If you do specify a data type, do not specify the length of CHAR, DECIMAL or MONEY fields. The length is determined by the width of the field in the SCREEN section.

FORMONLY removes the association and ties with the database itself. The normal rules enforced by the database server are not in effect, thereby placing all control at the application level.


Screen Arrays
DATABASE FORMONLY
SCREEN
CUSTOMER NO: [f01]
COMPANY: [f04]
NAME: [f02][f03]
ORDER INFORMATION
order Num order Dt PO Num Ship Dt Ship Chg Paid Dt
[f05] [f06] [f07] [f08] [f09] [f10]
[f05] [f06] [f07] [f08] [f09] [f10]
[f05] [f06] [f07] [f08] [f09] [f10]
[f05] [f06] [f07] [f08] [f09] [f10]
[f05] [f06] [f07] [f08] [f09] [f10]
END
ATTRIBUTES
f01 = FORMONLY.customer_num;
f02 = FORMONLY.fname;
f03 = FORMONLY.lname;
f04 = FORMONLY.company;
f05 = FORMONLY.order_num;
f06 = FORMONLY.order_date TYPE DATE, FORMAT = "MM/DD/YY";
f07 = FORMONLY.po_num;
f08 = FORMONLY.ship_date TYPE DATE, FORMAT = "MM/DD/YY";
f09 = FORMONLY.ship_charge TYPE MONEY, FORMAT = "$###.##";
f10 = FORMONLY.paid_date TYPE DATE, FORMAT = "MM/DD/YY";
END
INSTRUCTIONS
SCREEN RECORD sa_ord_rec[6] (order_num, order_date, po_num, ship_date,
ship_charge, paid_date);
END

Screen Arrays

The SCREEN RECORD statement in the INSTRUCTIONS section is used to define a screen array. The screen array number declared in the SCREEN RECORD statement must match the number of elements in the SCREEN section.

Referring to an Individual Field

In the example above, the **sa_ord_rec** screen array is defined with six elements and six fields in each element. The individual fields can be referenced as follows:

sa_ord_rec[element_number].field_name

DISPLAY ARRAY

```
DECLARE sel_cust_ord_curs CURSOR FOR
SELECT  orders.order_num, orders.order_date,
        orders.po_num, orders.ship_date,
        orders.ship_charge, orders.paid_date
FROM    orders
WHERE   orders.customer_num = p_customer_num
 . . .
LET row_cnt = 1
. . .
FOREACH sel_cust_ord_curs INTO ga_ord_rec[row_cnt].*
    LET row_cnt = row_cnt + 1
    IF ( row_cnt > 75 ) THEN
        EXIT FOREACH
    END IF
. . .
```


DISPLAY ARRAY

The DISPLAY ARRAY statement is used to display several rows of data within a screen array. It gives you the ability to scroll through the rows of data using the arrow keys. You are not limited to the size of the screen.

Displaying an array of data works in much the same way as displaying a single row. Rather than retrieving and storing a single row, you retrieve a set of rows and store them in a program array.

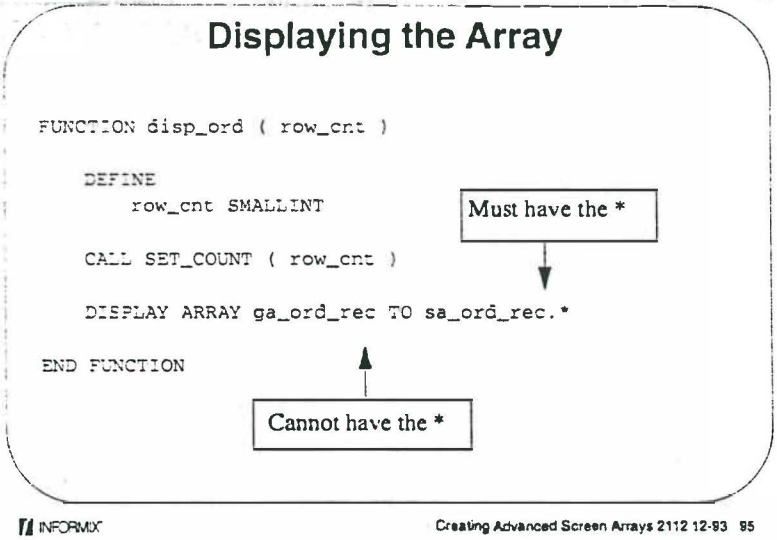
To retrieve data from the database:

- DECLARE a cursor
- OPEN a cursor
- FETCH the data into the program array
- Check the **sqlca.sqlcode** variable
- CLOSE the cursor

The difference between fetching a single row and multiple rows is:

- The data is fetched into a program array rather than a record variable.
- The FETCH is done in a loop.
- Since the data is stored in an array, you need to increment a variable that determines which element of the array is populated.

Note: In the IF-THEN statement block of code above, **75** indicates the size of the **ga_ord_rec** array defined in a GLOBALS section.



Displaying the Array

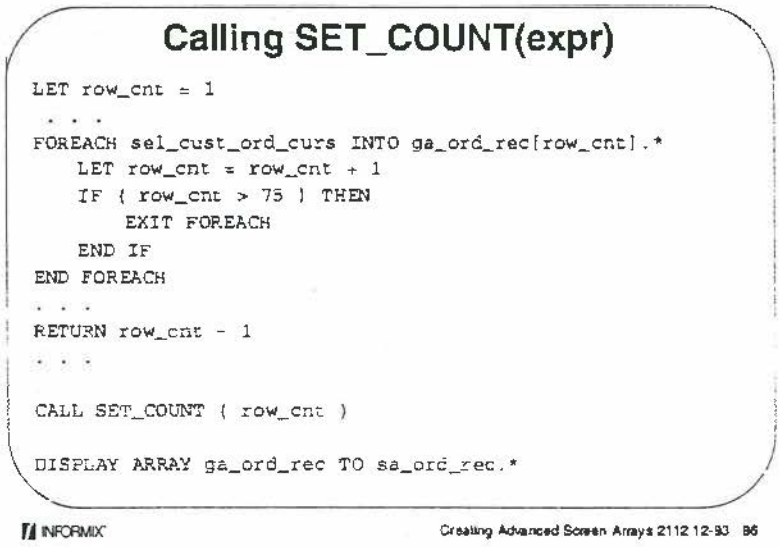
Displaying an array of data works much the same way as displaying a single row. Rather than the DISPLAY statement, you use the DISPLAY ARRAY statement.

SET_COUNT

You must use the SET_COUNT function to tell the DISPLAY ARRAY statement how many rows to expect.

Unlike the DISPLAY statement, the DISPLAY ARRAY statement turns control of the program over to the user.

- The user can use the arrow keys to move up and down in the array.
- In addition to using the arrow keys, the F3 key can be used to scroll down one page. The number of elements in the screen array is considered a page.
- The F4 key is used to scroll up a page.
- The DISPLAY ARRAY waits for the user to press the ACCEPT key or the INTERRUPT key signaling that they are ready to exit out of the array.
- The OPTIONS statement can be used to redefine which key performs these special tasks.

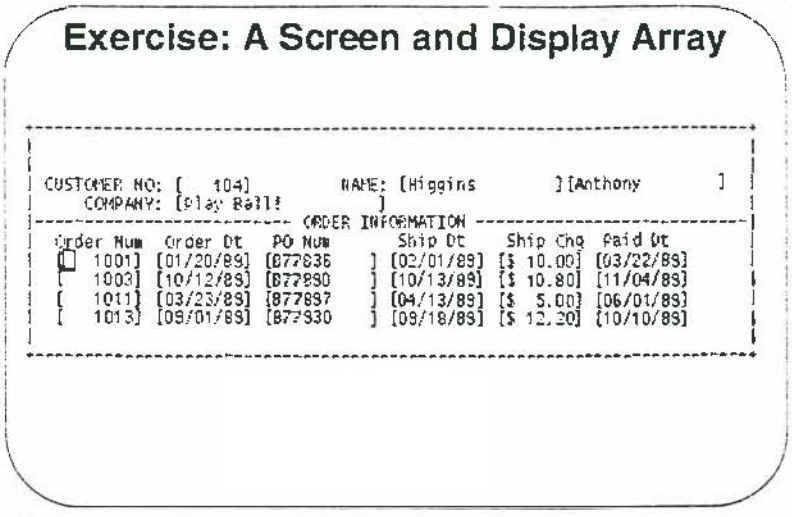


Calling SET_COUNT(expr)

You must use the SET_COUNT(expr) function to tell the DISPLAY ARRAY statement how many elements to expect. The argument to the SET_COUNT( ) function is usually not a constant; rather, the value is determined at the time the data is selected from the database.

In the example above, **row_cnt** is a variable used as an index into the array. The value of row_cnt is incremented as part of the FOREACH loop, so the actual number of rows fetched is row_cnt - 1.

Exercise: A Screen and Display Array



Exercise: A Screen and Display Array

Create a program that will display customer order information from the demonstration database. You should be in your $BASEDIR/src/exercise1 directory.

Step 1

Check out all source files from SCCS. For brevity, do not use SCCS for the remainder of the lab exercises.

Step 2

Edit your disp_win.per form file to be FORMONLY. Modify the form to include a screen array of four elements that displays the order number, order date, purchase order number, ship date, ship charge and paid date from the orders table.

Step 3

Edit your GLOBALS file globals.4gl to declare an array of order records with 75 elements.

Step 4

Edit your cust_ord.4gl module and add two functions: an order control (driver) function and an order display function. Name the order control function disp_ord_ctrl and the order display function disp_ord. Disp_ord_ctrl should accept one parameter, the customer number. It should call the function that selects the customer orders; if orders exist, it then calls the disp_ord function.

Step 5

Edit your svc_cust_ord.4gl module that contains your database access functions. Add a function to SELECT the order information from the orders table. Name this select function sel_cust_ord.

Step 6

Run the CCODE and FORM targets to compile your application and forms. Run your program.

Solutions

globals.4gl

```
DATABASE stores

GLOBALS

DEFINE

    ga_ord_rec ARRAY[75] OF RECORD
        order_num       LIKE orders.order_num,
        order_date      LIKE orders.order_date,
        po_num          LIKE orders.po_num,
        ship_date       LIKE orders.ship_date,
        ship_charge     LIKE orders.ship_charge,
        paid_date       LIKE orders.paid_date
    END RECORD

END GLOBALS
```

stub.4gl

MAIN

```
GLOBALS "globals.4gl"
MAIN

    DEFINE
    a_customer_num LIKE customer.customer_num

    IF ( NUM_ARGS ( ) != 1 ) THEN
        DISPLAY "USAGE: main.4ge <customer number>"
        DISPLAY ""
    ELSE
        LET a_customer_num = ARG_VAL(1)
        CALL cust_ctrl ( a_customer_num )
    END IF

END MAIN
```

cust_ord.4gl

cust_ctrl( )

```
GLOBALS "globals.4gl"

FUNCTION cust_ctrl ( p_customer_num )

    DEFINE
        cust_rec     RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

    p_customer_num LIKE customer.customer_num,

    sql_stat INTEGER

    WHENEVER ERROR STOP

    OPEN WINDOW cust_win AT 2,3 WITH FORM "disp_win"
    ATTRIBUTE ( BORDER )

    CALL sel_c                          um )
        RETURN                          .*

    IF ( sql_s
        ERROR  No customers found...."
        SLEEP 3
    ELSE
        CALL disp_cust ( cust_rec.* )

        CALL disp_ord_ctrl ( cust_rec.customer_num )

    END IF

    CLOSE WINDOW cust_win

END FUNCTION
```

disp_cust()

```
FUNCTION disp_cust ( p_cust_rec )

    DEFINE

        p_cust_rec   RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

    answer CHAR(1)

    DISPLAY BY NAME p_cust_rec.*

END FUNCTION
```

disp_ord_ctrl()

```
FUNCTION disp_ord_ctrl ( p_customer_num )

    DEFINE

        p_customer_num LIKE customer.customer_num,
        sql_stat INTEGER,
        row_cnt  SMALLINT,
        answer   CHAR(1)

    CALL sel_cust_ord ( p_customer_num )
        RETURNING sql_stat, row_cnt

    IF ( row_cnt = 0 ) THEN
        ERROR "Customer has NO orders"
        PROMPT "Press any key to continue...." FOR CHAR answer
    ELSE

        CALL disp_ord ( row_cnt )

    END IF

END FUNCTION
```

disp_ord()

```
FUNCTION disp_ord ( row_cnt )

    DEFINE

        row_cnt SMALLINT

    CALL SET_COUNT ( row_cnt )

    DISPLAY ARRAY pa_ord_rec TO sa_ord_rec.*

END FUNCTION
```

svc_cust_ord.4gl

sel_cust_rec( )

```
GLOBALS "globals.4gl"

FUNCTION sel_cust_rec ( p_customer_num )

    DEFINE

        cust_rec    RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

        p_customer_num LIKE customer.customer_num,

        sql_stat INTEGER

    WHENEVER ERROR CONTINUE

    SELECT customer.customer_num, customer.fname,
           customer.lname, customer.company
    INTO cust_rec.*
    FROM customer
    WHERE customer.customer_num = p_customer_num

    WHENEVER ERROR STOP

    LET sql_stat = SQLCA.SQLCODE

    RETURN sql_stat, cust_rec.*

END FUNCTION
```

sel_cust_ord()

```
FUNCTION sel_cust_ord ( p_customer_num )

    DEFINE

        p_customer_num LIKE customer.customer_num,
        sql_stat INTEGER,
        row_cnt  SMALLINT

    WHENEVER ERROR CONTINUE

    DECLARE sel_cust_ord_curs CURSOR FOR
     SELECT orders.order_num, orders.order_date,
            orders.po_num, orders.ship_date,
            orders.ship_charge, orders.paid_date
       FROM orders
      WHERE orders.customer_num = p_customer_num

    WHENEVER ERROR STOP

    LET sql_stat = SQLCA.SQLCODE

    LET row_cnt = 1

    IF ( NOT sql_stat ) THEN

        WHENEVER ERROR CONTINUE

        FOREACH sel_cust_ord_curs INTO ga_ord_rec[row_cnt].*

            WHENEVER ERROR STOP
            LET row_cnt = row_cnt + 1

            IF ( row_cnt > 75 ) THEN
                EXIT FOREACH
            END IF

        END FOREACH

    END IF

    RETURN sql_stat, row_cnt - 1

END FUNCTION
```

disp_win.per

```
DATABASE FORMONLY

SCREEN
{
 CUSTOMER NO: [f01    ]     NAME: [f03            ][f02           ]
     COMPANY: [f04              ]
-------------------------- ORDER INFORMATION --------------------
 Order Num  Order Dt    PO Num      Ship Dt    Ship Chg Paid Dt
 [f05     ] [f06     ] [f07       ] [f08     ] [f09     ] [f10     ]
 [f05     ] [f06     ] [f07       ] [f08     ] [f09     ] [f10     ]
 [f05     ] [f06     ] [f07       ] [f08     ] [f09     ] [f10     ]
 [f05     ] [f06     ] [f07       ] [f08     ] [f09     ] [f10     ]

}
END

ATTRIBUTES

    f01 = FORMONLY.customer_num;
    f02 = FORMONLY.fname;
    f03 = FORMONLY.lname;
    f04 = FORMONLY.company;
    f05 = FORMONLY.order_num;
    f06 = FORMONLY.order_date TYPE DATE, FORMAT = "MM/DD/YY";
    f07 = FORMONLY.po_num;
    f08 = FORMONLY.ship_date TYPE DATE, FORMAT = "MM/DD/YY";
    f09 = FORMONLY.ship_charge TYPE MONEY, FORMAT = "$###.##";
    f10 = FORMONLY.paid_date TYPE DATE, FORMAT = "MM/DD/YY";

END

INSTRUCTIONS

    SCREEN RECORD sa_ord_rec[4] ( order_num, order_date, po_num,
                                  ship_date,ship_charge,paid_date);

END
```

Syntax for INPUT ARRAY

```
INPUT ARRAY program_array WITHOUT DEFAULTS FROM screen_array.*
   HELP help_number
   ATTRIBUTE (attribute-list)
      BEFORE (INPUT|ROW|INSERT|DELETE|FIELD field_name)
          statement
      AFTER (INPUT|ROW|INSERT|DELETE|FIELD field_name)
          statement
      ON KEY ( key_list )
           Statement
           NEXT FIELD ( field_name | NEXT | PREVIOUS )
           CONTINUE INPUT
           EXIT INPUT

   END INPUT
```


Syntax for INPUT ARRAY

The INPUT ARRAY are required keywords. The *program_array* is the program array name. The *screen_array* is the name of the screen record.

WITHOUT DEFAULTS

Causes INFORMIX-4GL to display the values in the record array to the screen array. If you omit the WITHOUT DEFAULTS clause, INFORMIX-4GL initializes the first row of the screen array with the default values from the screen specification form. NULL values are assigned to variables with no default setting.

HELP *number*

This is used to specify a help message. The help message is generic to the INPUT ARRAY statement. The message associated with the help number should apply to the INPUT ARRAY statement as a whole, rather than a specific field.

ATTRIBUTES

Causes INFORMIX-4GL not to use the default attributes listed in the form specification file (or **syscolatt**).

BEFORE INPUT

Executes a group of statements before allowing the user to input any values.

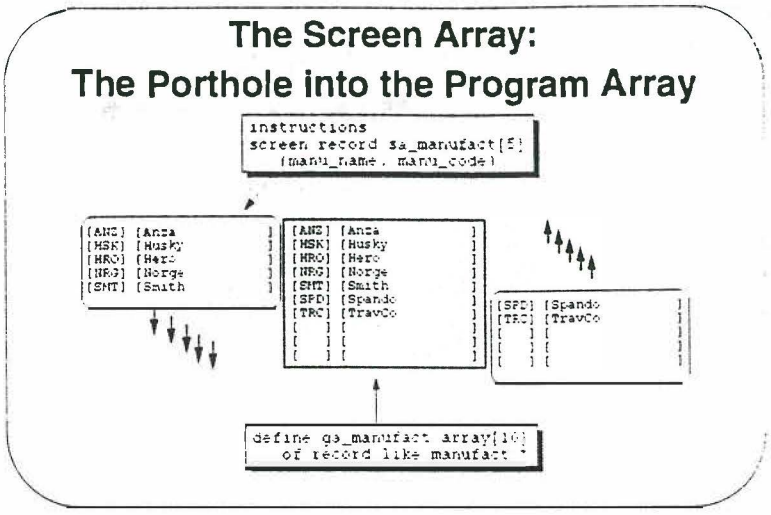
BEFORE ROW

Transfers control to an INFORMIX-4GL statement(s) when the cursor enters the first field of an element.

BEFORE INSERT

Transfers control to an INFORMIX-4GL statement(s) before adding a new line to the program array when the user presses the INSERT key (F1 by default) or before the first field of a new element.

BEFORE DELETE	Transfers control to an INFORMIX-4GL statement(s) before deleting the current line from the program array when the user presses the DELETE key (F2 by default).
BEFORE FIELD *field-name*	Transfers control to an INFORMIX-4GL statement(s) when the cursor enters the specified *field-name*.
AFTER INPUT	Transfers control to an INFORMIX-4GL statement(s) after the user has finished entering information. The statement(s) following the keywords will execute if the user presses the ACCEPT key or the INTERRUPT key.
AFTER ROW	Transfers control to an INFORMIX-4GL statement(s) when the cursor leaves an element, but before it arrives at a new element. This includes pressing the INSERT key (F1 by default).
AFTER INSERT	Transfers control to an INFORMIX-4GL statement(s) after 4GL inserts the new line to the program array and the user presses the INSERT key, or after the last field of a new element.
AFTER DELETE	Transfers control to an INFORMIX-4GL statement(s) after 4GL deletes the current line from the program array and the user presses the DELETE key (F2 by default).
AFTER FIELD *field-name*	Transfers control to an INFORMIX-4GL statement(s) when the cursor leaves the specified field.
ON KEY *(key-list)*	Transfers control to an INFORMIX-4GL statement(s) when a user presses the key or key sequence listed in the key-list.
NEXT FIELD *field-name*	Causes INFORMIX-4GL to move the cursor immediately to the specified field. This statement is used to bypass the natural cursor movement of the INPUT ARRAY statement.
CONTINUE INPUT	Returns control to the INPUT ARRAY statement and positions the cursor in the last occupied screen field of the current line.
EXIT INPUT	Causes INFORMIX-4GL to leave the INPUT ARRAY statement immediately.
END INPUT	Marks the end of the INPUT ARRAY statement. This statement is required if any of the clauses are used. The END INPUT statement *cannot* be used if no clauses are used.



The Screen Array: The Porthole into the Program Array

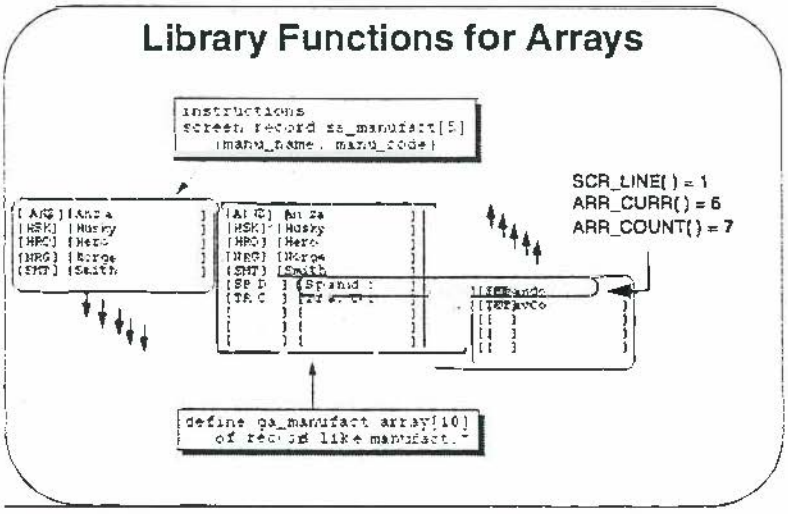
You can think of the screen array as a porthole that sits on top of the program array. In the example above, the user has the capability to enter up to ten rows (elements), but can only see five of those ten rows at any given time. By using the **arrow** keys, the user can move up and down in the program array. This is how the screen array and the program array work together.

The Size of the Arrays

When defining screen arrays and program arrays, the dimension of the program array must be greater than or equal to the dimension of the screen array.

Keeping Track of Your Position in the Arrays

It is important to be able to keep track of where you are in each of the arrays. For example, it is possible to be on the fifth element of the screen array and the sixth element of the program array. There must be a way to identify which element the cursor is currently on in the screen array and the program array.



Library Functions for Arrays

To aid the programmer in determining the location in the arrays, INFORMIX-4GL has the following library functions.

The SCR_LINE( ) Function Location in the Screen Array

The library function **SCR_LINE( )** is used to determine the cursor's position in the *screen* array. Below is an example of how it can be used:

```
DEFINE scrloc SMALLINT
LET scrloc = SCR_LINE( )
```

The variable **scrloc** will contain a number representing the element number of the *screen* array where the cursor currently resides.

The ARR_CURR( ) Function Location in the Program Array

The library function **ARR_CURR( )** is used to determine the cursor's position in the *program* array. It is used in much the same manner as **SCR_LINE( )**:

```
DEFINE prgloc SMALLINT
LET prgloc = ARR_CURR( )
```

The variable **prgloc** will contain a number representing the element number of the program array where the cursor currently resides.

The ARR_COUNT ( ) Function for Use with Arrays

The **ARR_COUNT( )** function is used to determine the last line the cursor occupied in the program array.

NOENTRY Fields

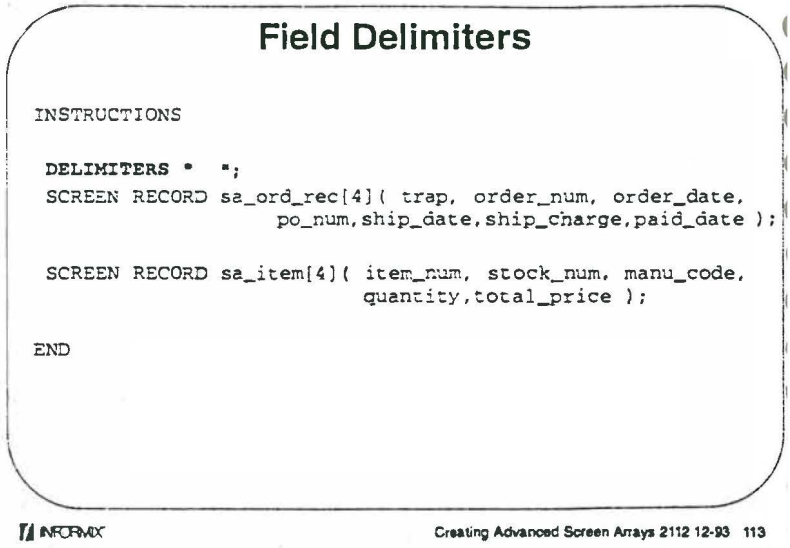
```
ATTRIBUTES
   a   = FORMONLY.crap;
   f01 = FORMONLY.customer_num, NOENTRY;
   f02 = FORMONLY.fname, NOENTRY;
   f03 = FORMONLY.lname, NOENTRY;
   f04 = FORMONLY.company, NOENTRY;
   f05 = FORMONLY.order_num, NOENTRY;
   f06 = FORMONLY.order_date TYPE DATE, FORMAT = "MM/DD/YY", NOENTRY;
   f07 = FORMONLY.po_num, NOENTRY;
   f08 = FORMONLY.ship_date TYPE DATE, FORMAT = "MM/DD/YY", NOENTRY;
   f09 = FORMONLY.ship_charge TYPE MONEY, FORMAT = "$###.##", NOENTRY;
   f10 = FORMONLY.paid_date TYPE DATE, FORMAT = "MM/DD/YY", NOENTRY;
   f11 = FORMONLY.item_num;
   f12 = FORMONLY.stock_num;
   f13 = FORMONLY.manu_code;
   f14 = FORMONLY.quantity;
   f15 = FORMONLY.total_price TYPE MONEY, FORMAT = "$*####.##", NOENTRY;
END
```


NOENTRY Fields

The NOENTRY attribute in a form specification file prevents data entry into the field during an INPUT or INPUT ARRAY statement. The NOENTRY attribute does not prevent data entry into a field during a CONSTRUCT statement.

Invisible Fields

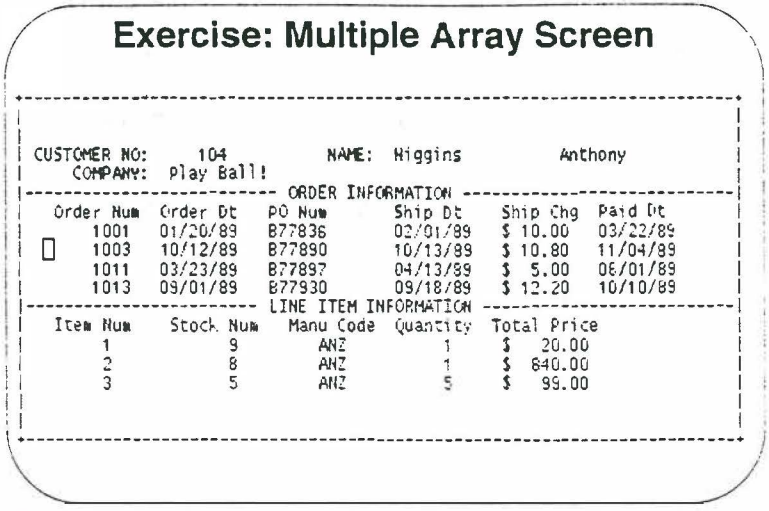
The INVISIBLE attribute, available with INFORMIX-4GL 4.1, causes 4GL not to display the value the user entered in the input field.



Field Delimiters

You can change the field delimiters that appear on the screen form from brackets to any other printable characters, including blank spaces. Form4gl requires brackets ([]) in the SCREEN section of the form specification file, regardless of the DELIMITERS instruction.

In the INSTRUCTIONS section, you define a symbol as both the beginning and ending delimiter. The delimiter symbols must be enclosed in quotation marks (").


Exercise: Multiple Array Screen
CUSTOMER NO: 104 NAME: Higgins Anthony
COMPANY: Play Ball!
ORDER INFORMATION
Order Num Order Dt PO Num Ship Dt Ship Chg Paid Dt
1001 01/20/89 B77836 02/01/89 $ 10.00 03/22/89
1003 10/12/89 B77890 10/13/89 $ 10.80 11/04/89
1011 03/23/89 B77897 04/13/89 $ 5.00 06/01/89
1013 09/01/89 B77930 09/18/89 $ 12.20 10/10/89
LINE ITEM INFORMATION
Item Num Stock Num Manu Code Quantity Total Price
1 9 ANZ 1 $ 20.00
2 8 ANZ 1 $ 840.00
3 5 ANZ 5 $ 99.00
INFORMIX
Creating Advanced Screen Arrays 2112 12-93 114

Exercise: Multiple Array Screen

The functional requirements of this program are as follows:

1. The program will display the customer order information from the demonstration database.
2. If the user reaches the end of the order array, position the cursor to *wrap* to the top of the array by placing a WHILE loop around your INPUT ARRAY statement block.
3. The item information for each order will be selected and displayed *as* the user moves the cursor through the customer orders.

   This will be accomplished by changing the order array from a DISPLAY ARRAY to an INPUT ARRAY to use the BEFORE ROW syntax (not available for DISPLAY ARRAY) to SELECT the item detail information as the user moves through the order array.

   You will need to use the **ARR_CUR** and **SCR_LINE** functions to display to the proper screen element.
4. To prevent the user from typing information into the orders array (INPUT ARRAY), you should declare a *trap* or *dummy* field as the first field of each element and all other fields as NOENTRY.

   If the user types a character into the trap field, clear the field or use the INVISIBLE attribute.
5. ON KEY (CONTROL-V) will allow the user to move the cursor *between* the order array and the item array to scroll through the item information.

You should be in your **$BASEDIR/src/exercise1** directory.

Step 1

Edit your **disp_win.per** form file. Modify the form as follows:

- Add a single character *trap* field at the first position of each array element in the order array. Define all other fields in the order array as NOENTRY.
- Add a screen array of four elements that displays the item number, stock number, manufacturers code, quantity, and total price from the items table.
- Change the delimiters to blanks " " to truly hide the *trap* field from the user.

Step 2

Edit your GLOBALS file to add the *trap* field to the array of order records. Declare an array of item records with 100 elements.

Step 3

Edit your **cust_ord.4gl** module. Edit your function that displayed the order information to include the program logic described earlier. Add the appropriate functions to display up to four items. In a real application, you may also consider mapping out the INSERT and DELETE keys to something unlikely (F25) to prevent the user from inserting or deleting an element from the INPUT ARRAY.

If the user presses a **Control-V**, switch to the items array (DISPLAY ARRAY) to allow the user to scroll through the remaining items (if there are more than four items).

Step 4

Edit your **svc_cust_ord.4gl** module that contains your database access functions. Add a function to SELECT the item information from the items table. Name the SELECT function **sel_ord_item**.

In addition, add a function that initializes the item array to NULL. Name this function **null_items**. For efficiency, initialize one array record to NULL and use the LET statement to copy the NULL record to the remaining array records.

Step 5

Run the CCODE and FORM targets to compile your application and form. At any time, run the PCODE target to use the Interactive Debugger.

Run your program.

Solutions

globals.4gl

```
DATABASE stores

GLOBALS

DEFINE

    ga_ord_rec ARRAY[75] OF RECORD
        trap            CHAR(1),
        order_num       LIKE orders.order_num,
        order_date      LIKE orders.order_date,
        po_num          LIKE orders.po_num,
        ship_date       LIKE orders.ship_date,
        ship_charge     LIKE orders.ship_charge,
        paid_date       LIKE orders.paid_date
    END RECORD,

    ga_item_ord ARRAY[100] OF RECORD
        item_num        LIKE items.item_num,
        stock_num       LIKE items.stock_num,
        manu_code       LIKE items.manu_code,
        quantity        LIKE items.quantity,
        total_price     LIKE items.total_price
    END RECORD

END GLOBALS
```

stub.4gl

main

```
GLOBALS "globals.4gl"

MAIN

    DEFINE
    a_customer_num LIKE customer.customer_num,
    sql_stat    INTEGER

    IF ( NUM_ARGS ( ) != 1 ) THEN
        DISPLAY "USAGE: main.4ge <customer number>"
        DISPLAY ""
    ELSE
        LET a_customer_num = ARG_VAL(1)
        CALL sel_ord_item_pre ( )
            RETURNING sql_stat

        IF ( NOT sql_stat ) THEN
            CALL cust_ctrl ( a_customer_num )
        ELSE
            DISPLAY "ERROR PREPARE/DECLARE ON items TABLE ",
            sql_stat
        END IF

    END IF

END MAIN
```

cust_ord.4gl

cust_ctrl( )

```
GLOBALS "globals.4gl"

FUNCTION cust_ctrl ( p_customer_num )

    DEFINE
        cust_rec     RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

    p_customer_num LIKE customer.customer_num,

    sql_stat INTEGER

    WHENEVER ERROR STOP

    OPEN WINDOW cust_win AT 2,3 WITH FORM "disp_win"
    ATTRIBUTE ( BORDER )

    CALL sel_cust_rec ( p_customer_num )
        RETURNING sql_stat, cust_rec.*

    IF ( sql_stat = NOTFOUND ) THEN
        ERROR "No customers found...."
        SLEEP 3
    ELSE
        CALL disp_cust ( cust_rec.* )

        CALL disp_ord_ctrl ( cust_rec.customer_num )

    END IF

    CLOSE WINDOW cust_win

END FUNCTION
```

disp_cust()

```
FUNCTION disp_cust ( p_cust_rec )

    DEFINE

        p_cust_rec   RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

    answer CHAR(1)

    DISPLAY BY NAME p_cust_rec.*

END FUNCTION
```

disp_ord_ctrl()

```
FUNCTION disp_ord_ctrl ( p_customer_num )

    DEFINE

        p_customer_num LIKE customer.customer_num,
        sql_stat INTEGER,
        row_cnt  SMALLINT,
        answer   CHAR(1)

    CALL sel_cust_ord ( p_customer_num )
        RETURNING sql_stat, row_cnt

    IF ( row_cnt = 0 ) THEN
        ERROR "Customer has NO orders"
        PROMPT "Press any key to continue...." FOR answer
    ELSE

    CALL disp_ord ( row_cnt )

    END IF

END FUNCTION
```

disp_ord()

```
FUNCTION disp_ord ( row_cnt )

    DEFINE

        sql_stat      INTEGER,
        item_row_cnt  SMALLINT,
        row_cnt       SMALLINT,
        cont_inp      SMALLINT,
        cur_row       SMALLINT,
        scr_row       SMALLINT

    CALL SET_COUNT ( row_cnt )

    LET cont_inp = TRUE

    WHILE ( cont_inp = TRUE )

        INPUT ARRAY ga_ord_rec WITHOUT DEFAULTS FROM sa_ord_rec.*

            AFTER FIELD trap

                LET ga_ord_rec[cur_row].trap = NULL
                DISPLAY "" TO sa_ord_rec[scr_row].trap

            BEFORE ROW
                LET cur_row = ARR_CURR ( )
                LET scr_row = SCR_LINE ( )
                IF ( cur_row = row_cnt + 1 ) THEN
                    LET cont_inp = TRUE
                    EXIT INPUT
                ELSE
                    LET cont_inp = FALSE
                    CALL proc_items ( cur_row )
                        RETURNING sql_stat, item_row_cnt
                END IF

            ON KEY ( CONTROL-V )
                CALL disp_array_items ( item_row_cnt )

        END INPUT

    END WHILE

END FUNCTION
```

proc_items()

```
FUNCTION proc_items ( p_cur_row )

    DEFINE

        sql_stat INTEGER,

        p_cur_row SMALLINT,
        item_row_cnt SMALLINT

    CALL sel_ord_item ( p_cur_row
        RETURNING sql_stat, item_row_cnt

    CALL disp_four_items ( )

    RETURN sql_stat, item_row_cnt

END FUNCTION
```

disp_four_items()

```
FUNCTION disp_four_items ( )

    DEFINE
        i SMALLINT

    FOR i = 1 TO 4
        DISPLAY ga_item_ord[i].* TO sa_item[i].*
    END FOR

END FUNCTION
```

disp_array_items()

```
FUNCTION disp_array_items ( p_item_row_cnt )

    DEFINE
        p_item_row_cnt SMALLINT

    CALL SET_COUNT ( p_item_row_cnt

    DISPLAY ARRAY ga_item_ord TO sa_item.*

        ON KEY ( CONTROL-V
            EXIT DISPLAY

    END DISPLAY

END FUNCTION
```

svc_cust_ord.4gl

sel_cust_rec( )

```
GLOBALS "globals.4gl"

FUNCTION sel_cust_rec ( p_customer_num )

    DEFINE

        cust_rec    RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

        p_customer_num LIKE customer.customer_num,

        sql_stat INTEGER

    WHENEVER ERROR CONTINUE

    SELECT customer.customer_num, customer.fname,
           customer.lname, customer.company
    INTO cust_rec.*
    FROM customer
    WHERE customer.customer_num = p_customer_num

    WHENEVER ERROR STOP

    LET sql_stat = SQLCA.SQLCODE

    RETURN sql_stat, cust_rec.*

END FUNCTION
```

sel_cust_ord( )

```
FUNCTION sel_cust_ord ( p_customer_num )

    DEFINE

        p_customer_num  LIKE customer.customer_num,
        sql_stat        INTEGER,
        row_cnt         SMALLINT

    WHENEVER ERROR CONTINUE

    DECLARE sel_cust_ord_curs CURSOR FOR
     SELECT * *, orders.order_num, orders.order_date,
            orders.po_num, orders.ship_date,
            orders.ship_charge, orders.paid_date
       FROM orders
      WHERE orders.customer_num = p_customer_num

    WHENEVER ERROR STOP

    LET sql_stat = SQLCA.SQLCODE

    LET row_cnt = 1

    IF ( NOT sql_stat ) THEN

        WHENEVER ERROR CONTINUE

        FOREACH sel_cust_ord_curs INTO ga_ord_rec[row_cnt].*

            WHENEVER ERROR STOP
            LET row_cnt = row_cnt + 1

            IF ( row_cnt > 75 ) THEN
                EXIT FOREACH
            END IF

        END FOREACH

    END IF

    RETURN sql_stat, row_cnt - 1

END FUNCTION
```

sel_ord_item_pre()

```
FUNCTION sel_ord_item_pre ( )

    DEFINE

        sql_text CHAR(250),
        sql_stat INTEGER

    LET sql_text = " SELECT items.item_num, items.stock_num, ",
                           " items.manu_code, items.quantity, ",
                           " items.total_price ",
                       " FROM items ",
                      " WHERE items.order_num = ? "

    WHENEVER ERROR CONTINUE

    PREPARE sel_item_stmt FROM sql_text

    WHENEVER ERROR STOP

    LET sql_stat = SQLCA.SQLCODE

    IF ( NOT sql_stat ) THEN

        WHENEVER ERROR CONTINUE

        DECLARE sel_item_curs CURSOR FOR sel_item_stmt

        WHENEVER ERROR STOP

         LET sql_stat = SQLCA.SQLCODE

    END IF

    RETURN sql_stat

END FUNCTION
```

sel_ord_item()

```
FUNCTION sel_ord_item ( p_cur_row )

    DEFINE

        sql_stat INTEGER,

        p_cur_row SMALLINT,
        row_cnt SMALLINT

    CALL null_items ( )

    WHENEVER ERROR CONTINUE

    OPEN sel_item_curs USING ga_ord_rec[p_cur_row].order_num

    WHENEVER ERROR STOP

    LET sql_stat = SQLCA.SQLCODE

    LET row_cnt = 1

    WHILE ( ( NOT sql_stat ) AND ( row_cnt <= 100 ) )

        WHENEVER ERROR CONTINUE

        FETCH sel_item_curs INTO ga_item_ord[row_cnt].*

        WHENEVER ERROR STOP

        LET sql_stat = SQLCA.SQLCODE

        IF ( NOT sql_stat ) THEN
            LET row_cnt = row_cnt + 1
        END IF

    END WHILE

    IF ( sql_stat = 100 ) THEN
        LET sql_stat = 0
    END IF

    RETURN sql_stat, row_cnt - 1

END FUNCTION
```

null_items()

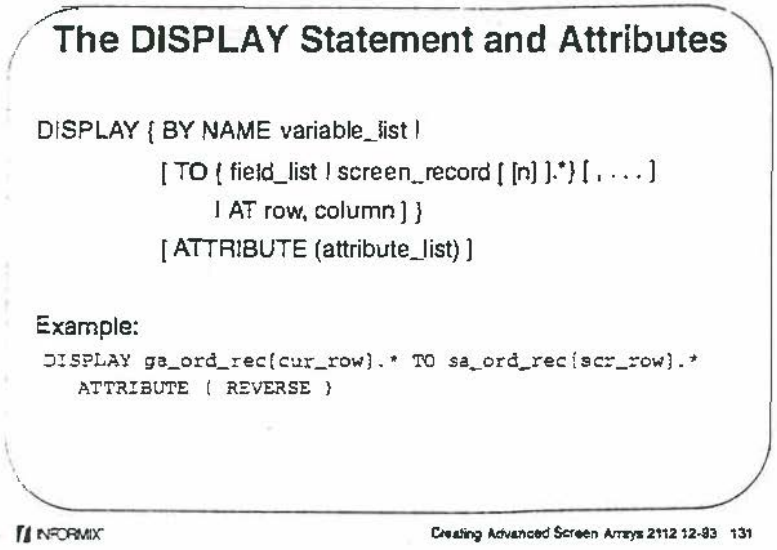
```
FUNCTION null_items ( )

    DEFINE
        i SMALLINT

    INITIALIZE ga_item_ord[1].* TO NULL

    FOR i = 2 TO 100
        LET ga_item_ord[i].* = ga_item_ord[1].*
    END FOR

END FUNCTION
```

disp_win.per

```
DATABASE FORMONLY

SCREEN
{
 CUSTOMER NO: [f01    ]      NAME: [f03            ][f02           ]
     COMPANY: [f04                ]
------------------------- ORDER INFORMATION -------------------
 Order Num  Order Dt     PO Num       Ship Dt     Ship Chg Paid Dt
[a|f05     ] [f06      ] [f07       ] [f08      ] [f09     ][f10     ]
[a|f05     ] [f06      ] [f07       ] [f08      ] [f09     ][f10     ]
[a|f05     ] [f06      ] [f07       ] [f08      ] [f09     ][f10     ]
[a|f05     ] [f06      ] [f07       ] [f08      ] [f09     ][f10     ]
------------------------ LINE ITEM INFORMATION ------------------
 Item Num      Stock Num     Manu Code    Quantity      Total Price
 [f11  ]       [f12  ]       [f13]        [f14 ]        [f15        ]
 [f11  ]       [f12  ]       [f13]        [f14 ]        [f15        ]
 [f11  ]       [f12  ]       [f13]        [f14 ]        [f15        ]
 [f11  ]       [f12  ]       [f13]        [f14 ]        [f15        ]
}
END

ATTRIBUTES

    a   = FORMONLY.trap;
    f01 = FORMONLY.customer_num, NOENTRY;
    f02 = FORMONLY.fname, NOENTRY;
    f03 = FORMONLY.lname, NOENTRY;
    f04 = FORMONLY.company, NOENTRY;
    f05 = FORMONLY.order_num, NOENTRY;
    f06 = FORMONLY.order_date TYPE DATE, FORMAT = "MM/DD/YY",
                NOENTRY;
    f07 = FORMONLY.po_num, NOENTRY;
    f08 = FORMONLY.ship_date TYPE DATE, FORMAT = "MM/DD/YY",
                NOENTRY;
    f09 = FORMONLY.ship_charge TYPE MONEY, FORMAT = "$###.##",
                NOENTRY;
    f10 = FORMONLY.paid_date TYPE DATE, FORMAT = "MM/DD/YY",
                NOENTRY;
    f11 = FORMONLY.item_num;
    f12 = FORMONLY.stock_num;
    f13 = FORMONLY.manu_code;
    f14 = FORMONLY.quantity;
    f15 = FORMONLY.total_price TYPE MONEY, FORMAT = "$######.##";
END
```

disp_win.per (cont.)

```
INSTRUCTIONS

    DELIMITERS "  ";

    SCREEN RECORD sa_ord_rec[4] ( trap, order_num, order_date,
                        po_num,ship_date,ship_charge,paid_date);

    SCREEN RECORD sa_item[4] ( item_num, stock_num, manu_code,
                               quantity, total_price );

END
```

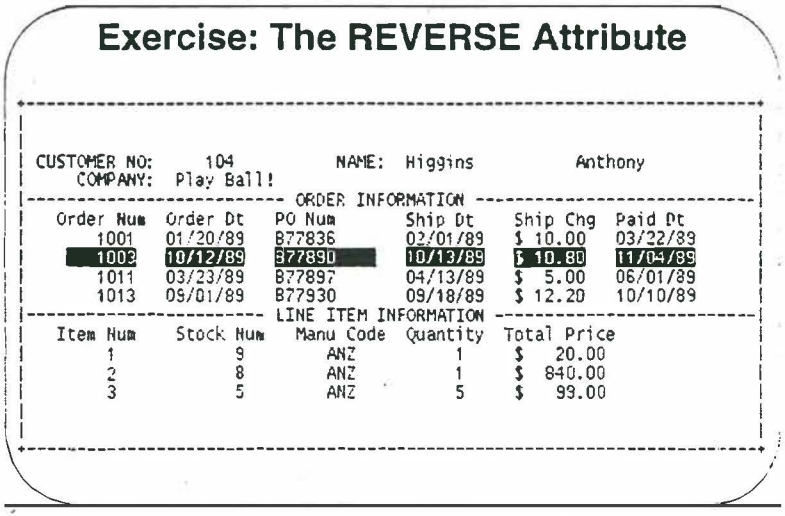

The DISPLAY Statement and Attributes
DISPLAY { BY NAME variable_list |
[TO { field_list | screen_record [[n]].* } [, . . .]
| AT row, column] }
[ATTRIBUTE (attribute_list)]
Example:
DISPLAY ga_ord_rec[cur_row].* TO sa_ord_rec[scr_row].*
ATTRIBUTE (REVERSE)
INFORMIX
Creating Advanced Screen Arrays 2112 12-93 131

The DISPLAY Statement and Attributes

The DISPLAY statement is used to display data values on the screen. You can specify screen display attributes in the ATTRIBUTE clause. The REVERSE, BLINK and UNDERLINE keywords can appear in the ATTRIBUTE clause.

If you use the ATTRIBUTE clause, none of the default attributes listed in the syscolatt table or in the form specification file for fields apply.

Exercise: The REVERSE Attribute


CUSTOMER NO: 104 NAME: Higgins Anthony
COMPANY: Play Ball!
ORDER INFORMATION
Order Num Order Dt PO Num Ship Dt Ship Chg Paid Dt
1001 01/20/89 B77836 02/01/89 $ 10.00 03/22/89
1002 10/12/89 B77890 10/13/89 $ 10.80 11/04/89
1011 03/23/89 B77897 04/13/89 $ 5.00 06/01/89
1013 09/01/89 B77930 09/18/89 $ 12.20 10/10/89
LINE ITEM INFORMATION
Item Num Stock Num Manu Code Quantity Total Price
1 9 ANZ 1 $ 20.00
2 8 ANZ 1 $ 840.00
3 5 ANZ 5 $ 99.00

Exercise: The REVERSE Attribute

Modify your program so that the current line is highlighted as the user moves the cursor through the order array. Before the current row, display the order record with a REVERSE attribute and after the current row, display the row without the REVERSE attribute.

Step 1

Edit your **disp_ord** function to include the **BEFORE ROW** and **AFTER ROW** logic within your INPUT ARRAY statement block.

Step 2

Run the CCODE target to compile your application. Run your program.

Solutions

cust_ord.4gl

disp_ord()

```
...
FUNCTION disp_ord ( row_cnt )

    DEFINE

        sql_stat     INTEGER,
        item_row_cnt SMALLINT,
        row_cnt      SMALLINT,
        cont_inp     SMALLINT,
        cur_row      SMALLINT,
        scr_row      SMALLINT

    CALL SET_COUNT ( row_cnt )

    LET cont_inp = TRUE

    WHILE ( cont_inp = TRUE )

        INPUT ARRAY ga_ord_rec WITHOUT DEFAULTS FROM sa_ord_rec.*

            AFTER FIELD trap

                LET ga_ord_rec[cur_row].trap = NULL
                DISPLAY "" TO sa_ord_rec[scr_row].trap

            AFTER ROW

                LET cur_row = ARR_CURR ( )
                LET scr_row = SCR_LINE ( )

                DISPLAY ga_ord_rec[cur_row].* TO
                        sa_ord_rec[scr_row].*

            BEFORE ROW

                LET cur_row = ARR_CURR ( )
                IF ( cur_row = row_cnt + 1 ) THEN
                    LET cont_inp = TRUE
                    EXIT INPUT
                ELSE
                    LET scr_row = SCR_LINE ( )
                    DISPLAY ga_ord_rec[cur_row].* TO
                            sa_ord_rec[scr_row].*
                        ATTRIBUTE ( REVERSE )
                    LET cont_inp = FALSE
```

```
                CALL proc_items ( cur_row )
                    RETURNING sql_stat, item_row_cnt
            END IF

        ON KEY ( CONTROL-V )
            CALL disp_array_items ( item_row_cnt )

    END INPUT

END WHILE

END FUNCTION
```

INFORMIX®

Chapter 7

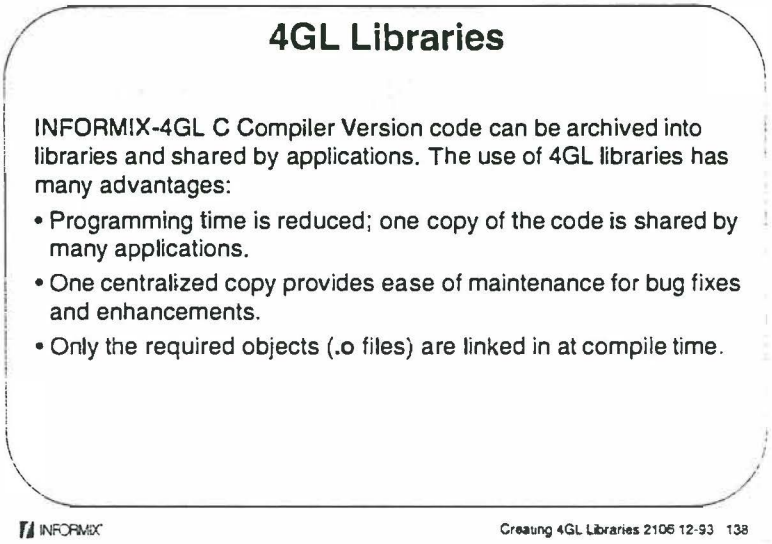
Creating 4GL Libraries

Objectives

At the end of this module, you will be able to:

- Understand the benefits of centralized 4GL libraries
- Use the UNIX archive command to create INFORMIX-4GL libraries
- Create pseudo libraries for INFORMIX-4GL Rapid Development System
- Understand the different types of error handling
- Create a generic error function library file to be used with the WHENEVER ERROR CALL statement
- Create custom error messages

© 1993 Informix Software, Inc

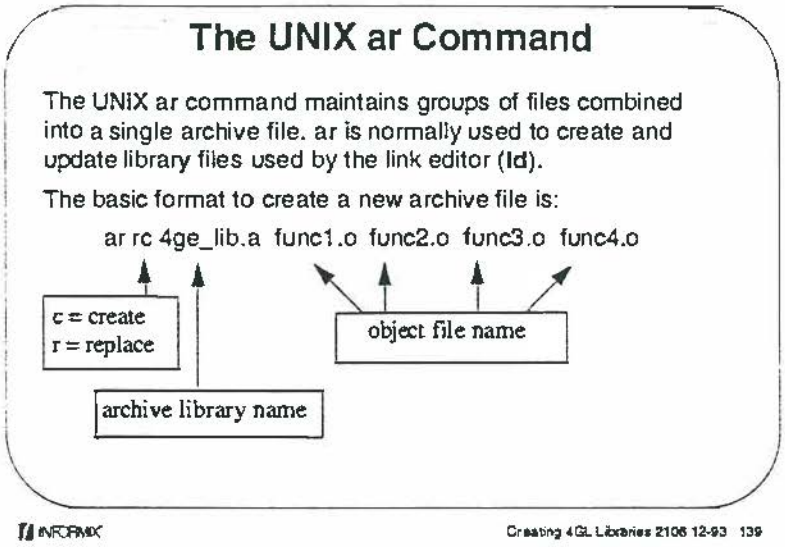

4GL Libraries
INFORMIX-4GL C Compiler Version code can be archived into libraries and shared by applications. The use of 4GL libraries has many advantages:
• Programming time is reduced; one copy of the code is shared by many applications.
• One centralized copy provides ease of maintenance for bug fixes and enhancements.
• Only the required objects (.o files) are linked in at compile time.
INFORMIX
Creating 4GL Libraries 2106 12-93 138

4GL Libraries

The UNIX archive utility ar can be used to create libraries of common INFORMIX-4GL code. A few examples of common 4GL code includes:

- validation routines
- initialization routines
- look-up windows
- error and exception handling routines

Common code across applications is usually copied and re-used from one development project to the next. However, multiple copies of common code can present problems. If the source code has a bug, then all copies of the code must be fixed. If code enhancements are desired, then all copies of the code must be updated.


The UNIX ar Command
The UNIX ar command maintains groups of files combined into a single archive file. ar is normally used to create and update library files used by the link editor (ld).
The basic format to create a new archive file is:
ar rc 4ge_lib.a func1.o func2.o func3.o func4.o
c = create
r = replace
object file name
archive library name
INFORMIX
Creating 4GL Libraries 2106 12-93 139

The UNIX ar Command

The UNIX **ar** command can be used to group files into a single library archive file. An archive file is made up of a one or more *member* files and header information for each file. The library archive file can then be used by the link editor (**ld**).

The basic command to create a new archive file to use as a 4GL library is:

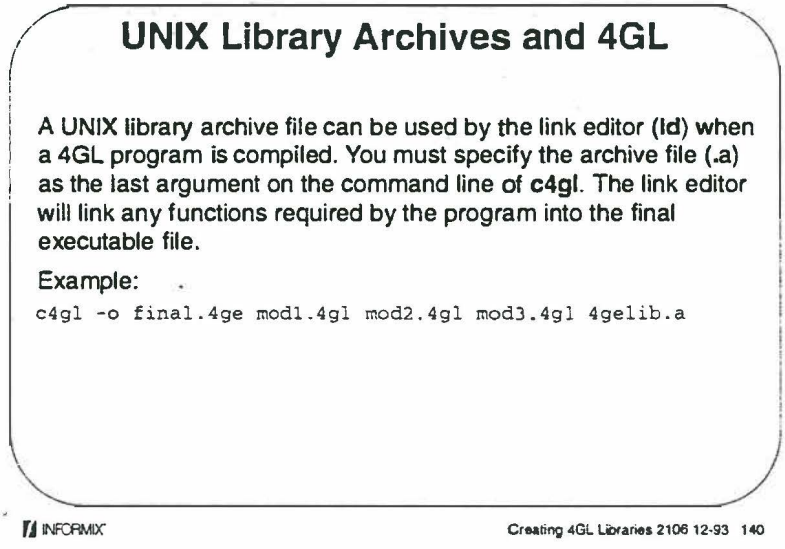
```
ar rc [library archive name] [4GL object file names]
```

The library archive name is an arbitrary name with a .a extension. The object file name is the 4GL object file name with a .o extension.

Since **ar** is a UNIX command, you should refer to your UNIX system manual for complete system specific information.

Makefiles

You can define dependencies in your **makefile** to create your .4gi and .4ge 4GL libraries.


UNIX Library Archives and 4GL
A UNIX library archive file can be used by the link editor (ld) when a 4GL program is compiled. You must specify the archive file (.a) as the last argument on the command line of c4gl. The link editor will link any functions required by the program into the final executable file.
Example:
c4gl -o final.4ge mod1.4gl mod2.4gl mod3.4gl 4gelib.a
INFORMIX
Creating 4GL Libraries 2106 12-93 140

UNIX Library Archives and 4GL

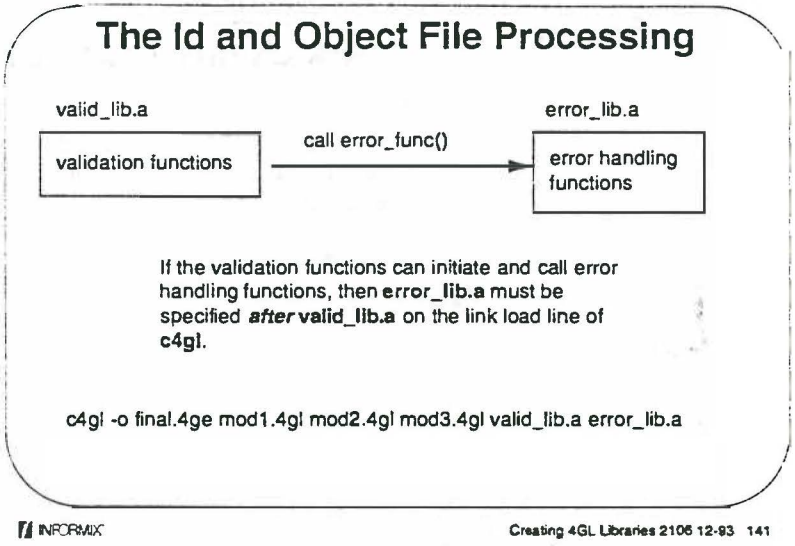
A UNIX library archive file (.a) can be used on the **c4gl** command line to link objects from the archive file into the final 4GL executable. The archive library file must be specified as the last object on the command line.

Makefiles

You must update your .4ge target in your dependency definition to include the library file on the command line.

The Linker

Special consideration must be taken if you have multiple libraries and library functions call library functions. The order of the library files specified on the link load line is very important. The link editor is discussed in more detail on the following page.


The ld and Object File Processing
valid_lib.a
validation functions
call error_func()
error_lib.a
error handling functions
If the validation functions can initiate and call error handling functions, then error_lib.a must be specified after valid_lib.a on the link load line of c4gl.
c4gl -o final.4ge mod1.4gl mod2.4gl mod3.4gl valid_lib.a error_lib.a
INFORMIX
Creating 4GL Libraries 2106 12-93 141

The ld and Object File Processing

The order of the library files on the command line is very important if you have library functions which call library functions in separate archive files.

Note:

A brief explanation of object file processing by the link editor (ld) is below. You should refer to your UNIX system manual for complete information.

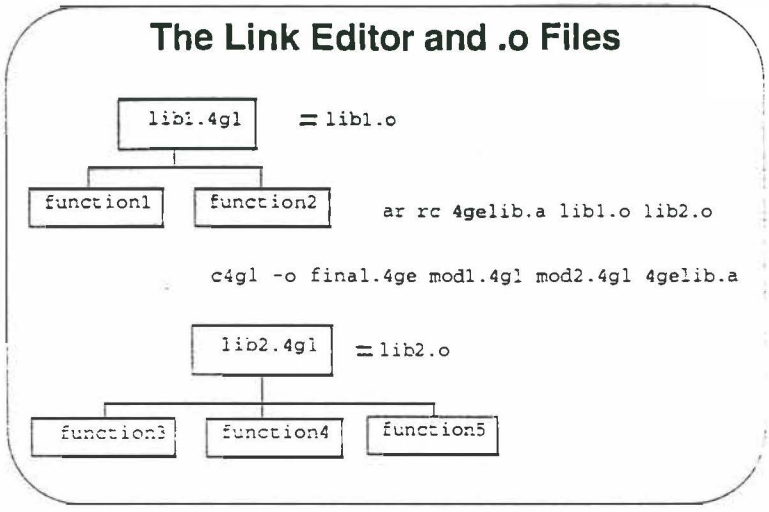
Files specified on the command line are processed in the order listed. Information is extracted and concatenated from each file to form the output file. The specific processing on a given file depends on if the file is a simple object file or a library archive.

Simple object files (.o) are concatenated to the output as they are encountered on the link load line. Each library archive file (.a) is searched *exactly once* as it is encountered on the command line. Only the archive functions matching an unresolved external reference are extracted and concatenated to the output.

If a member of an archive references a function defined by another member, *the member making the reference must appear before the member containing the definition.*

If there remains an unresolved external reference, you will receive an error at the final stage of compilation:

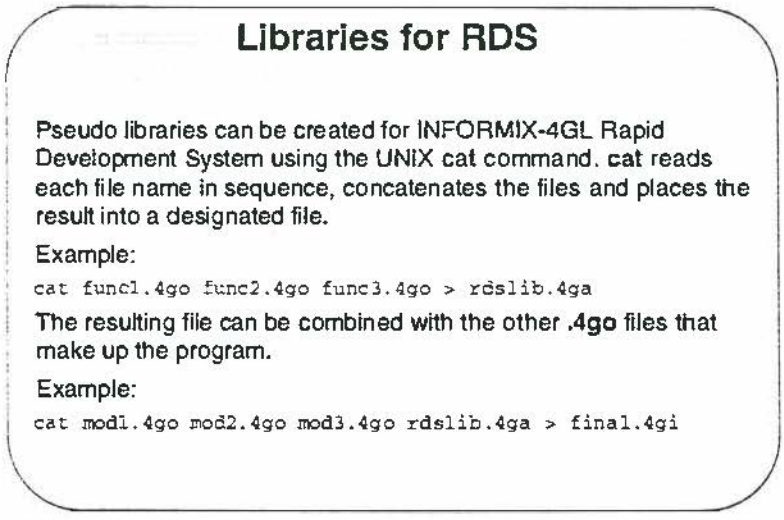
"Undefined Symbol *function_name ()*"



The Link Editor and .o Files

The smallest component that can be linked at compile time from an archive library file (**.a**) is a **.o** file. In the example above, if **mod1.4gl** calls **function1** only, then the entire **lib1.o** will be included in **final.4ge**. The **lib2.o** is not included in **final.4ge**.

If a ***file*.o** contains five functions and your program requires only one of the functions, all five will be linked into the final executable. In the example above, if **mod1.4gl** and **mod2.4gl** call **function4** in **lib2.4gl** and never call **function3** and **function5**, the entire **lib2.o** will be included in **final.4ge**. This should be a primary consideration when deciding how to implement 4GL libraries. It is not uncommon to have many archive library files (.a).


Libraries for RDS
Pseudo libraries can be created for INFORMIX-4GL Rapid Development System using the UNIX cat command. cat reads each file name in sequence, concatenates the files and places the result into a designated file.
Example:
cat func1.4go func2.4go func3.4go > rdslib.4ga
The resulting file can be combined with the other .4go files that make up the program.
Example:
cat mod1.4go mod2.4go mod3.4go rdslib.4ga > final.4gi
INFORMIX
Creating 4GL Libraries 2106 12-93 143

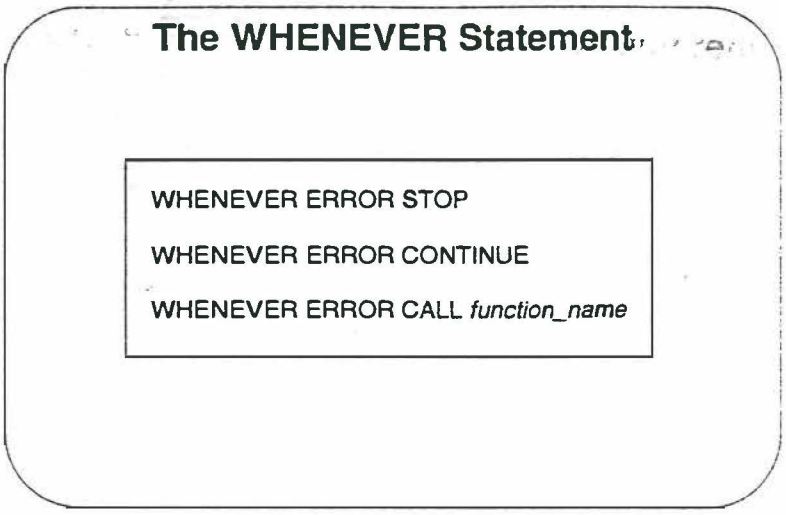
Libraries for RDS

You can create pseudo libraries using the UNIX **cat** command to concatenate **.4go** files into a single *library* file. The *library* file can then be concatenated with the other **.4go** files that make up the program.

Makefiles

You must update your **.4gi** target in your dependency definition to include the library file on the command line.

Since **cat** is a UNIX command, you should refer to your UNIX system manual for complete information.



The WHENEVER Statement

The WHENEVER statement is used to change what a program does when an error occurs. A WHENEVER statement can instruct a program to:

- Stop when an error occurs
- Continue, despite the error
- Call a function when an error occurs

WHENEVER ERROR STOP

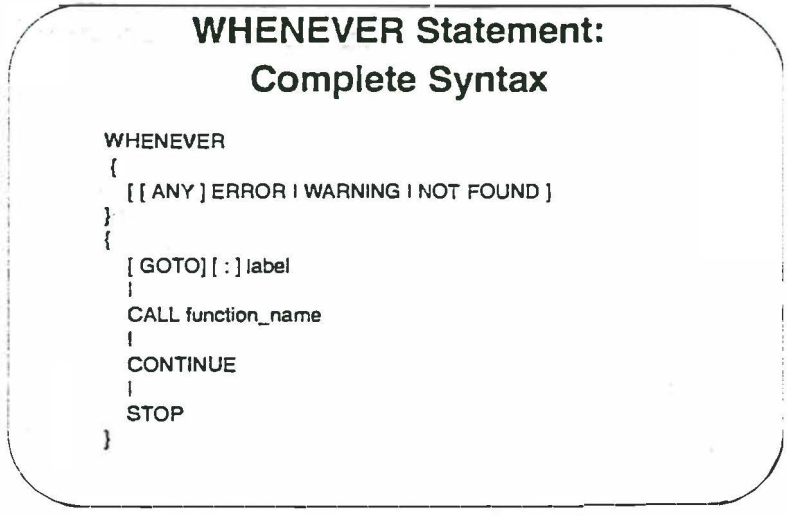
WHENEVER ERROR STOP prints an error message and then stops the program execution. This is a program's default condition.

WHENEVER ERROR CONTINUE

WHENEVER ERROR CONTINUE keeps the program from stopping when a run-time error occurs. When WHENEVER ERROR CONTINUE is in effect, it is very important for the programmer to provide error checking after every statement that could cause an error.

WHENEVER ERROR CALL *function-name*

WHENEVER ERROR CALL function-name calls the specified function when a run-time error occurs. Use this form when you want to create your own error checking routines. Note that this function cannot be passed parameters.



WHENEVER Statement: Complete Syntax

The three most common forms of the WHENEVER statement were explained on the last page. This slide presents the complete syntax of the WHENEVER statement.

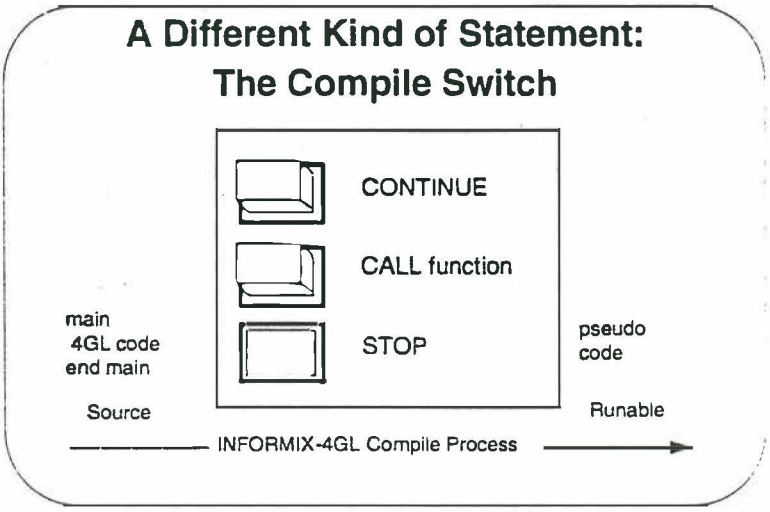
The ANY keyword and WHENEVER WARNING clause are described below. The NOT FOUND clause is not discussed in this module. Use of the GOTO clause is not recommended.

WHENEVER ANY

INFORMIX-4GL Version 4.1 introduced the ANY keyword to the WHENEVER statement. If you specify WHENEVER ANY, INFORMIX-4GL sets the value of the status variable after evaluating an expression, in addition to doing so after checking SQL statements, screen I/O statements, and the VALIDATE statement.

WHENEVER WARNING

WHENEVER WARNING functions like WHENEVER ERROR except that it tests after each 4GL statement whether or not SQLAWARN[1] has been set to W.



A Different Kind of Statement: The Compile Switch

Most INFORMIX-4GL statements are run-time instructions to the program. For example, the MENU statement is a run-time instruction to display a certain type of menu structure. The WHENEVER statement is very different from other INFORMIX-4GL statements; it is called a *compile switch*.

A Compile Switch

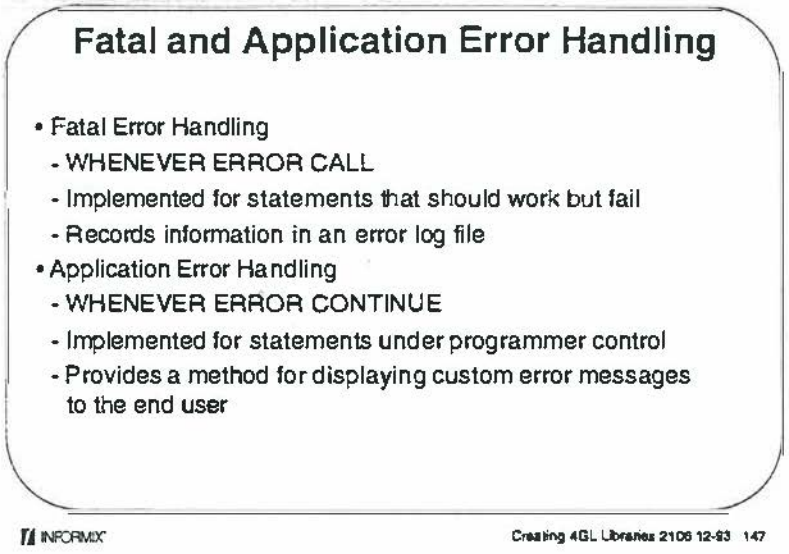
A compile switch, as its name implies, is an instruction to the compiler, *not* the program. The switch tells the compiler what to do with statements that can cause errors. For example, if left in the default position of STOP, the compiler inserts code into your program to surround all statements that could cause errors. It is this switch's job to know which statements those are. When an error is encountered at run-time, that extra code will display an error message and stop the program.

Changing the Switch The WHENEVER Statement

When you put a WHENEVER statement into your 4GL program, you are actually choosing a different option for the compile switch. In other words, you are telling the compiler what to do with all those statements that could cause errors.

Scope of WHENEVER

The WHENEVER statement has module scope. For each module, the compiler uses the last setting encountered sequentially for the WHENEVER statement as it compiles your code.


Fatal and Application Error Handling
• Fatal Error Handling
- WHENEVER ERROR CALL
- Implemented for statements that should work but fail
- Records information in an error log file
• Application Error Handling
- WHENEVER ERROR CONTINUE
- Implemented for statements under programmer control
- Provides a method for displaying custom error messages to the end user
INFORMIX
Creating 4GL Libraries 2106 12-93 147

Fatal and Application Error Handling

Error handling routines are an important part of developing applications. Error handling routines, whether robust or simple, usually share common code and are an excellent candidate for a 4GL library.

There is a distinction between recoverable application errors and non-recoverable fatal errors. It is important to understand these distinctions in order to implement efficient and robust error handling routines.

Fatal error handling should be implemented for statements that should work but fail, and are not under the control of the end user. An example of a fatal error might include:

- -1110 Form file not found.
- -1136 Window is too large to fit on the screen.

Application error handling should be implemented for statements that fail but are under programmer or user control. An example of an application error might include:

- -378 Record currently locked by another user.
- -239 Could not insert new row - duplicate value in a UNIQUE INDEX column.

SQLCA.SQLCODE and Status

It is recommended that you refer to the SQL error code in the SQL Communications Area for the following reasons:

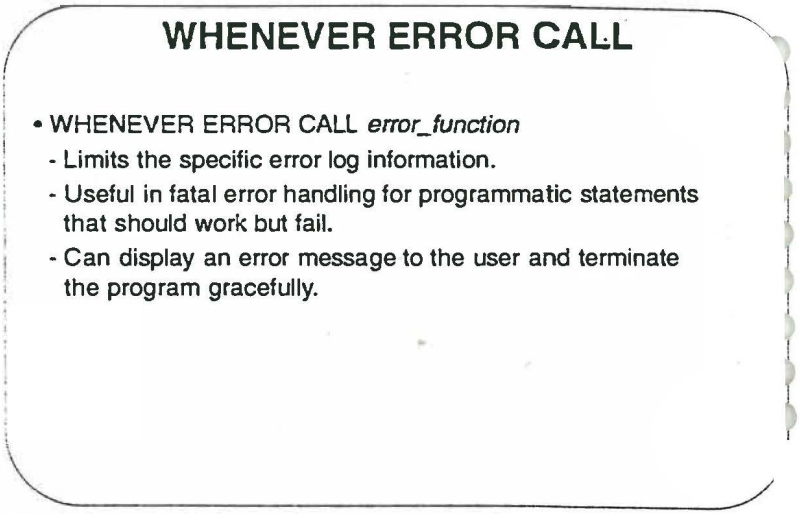
- The STATUS variable and SQLCA.SQLCODE are not the same.
- STATUS records the results of SQL statement execution and INFORMIX-4GL form related statements. Therefore, an SQL error code may not remain in STATUS as expected following the SQL statement.
- The STATUS variable is evaluated differently between versions of INFORMIX-4GL.

SQLCA.SQLCODE and Status

The SQLCA.SQLCODE records the results of SQL statement execution (except for the DECLARE statement). The STATUS variable records the results of SQL statement execution and the results of form related INFORMIX-4GL statements. Therefore, it is not recommended to assume that the SQL error code will remain in STATUS following the SQL statement.

The STATUS variable is evaluated differently between versions of INFORMIX-4GL. When determining the portability of your application, you should carefully consider when to use the STATUS variable.

Tip: If you choose to use STATUS, a recommended technique to ensure that the correct value of STATUS is always available is to save the value of STATUS in a local variable immediately following the SQL statement.


WHENEVER ERROR CALL
• WHENEVER ERROR CALL error_function
- Limits the specific error log information.
- Useful in fatal error handling for programmatic statements that should work but fail.
- Can display an error message to the user and terminate the program gracefully.
INFORMIX
Creating 4GL Libraries 2106 12-93 149

WHENEVER ERROR CALL

The WHENEVER ERROR CALL statement limits the specific log information that can be supplied about an error. You cannot pass parameters to the error logging function. If variations of error handling are required, then multiple *error_function(s)* are necessary. Multiple *error_function* routines can become cumbersome and difficult to manage.

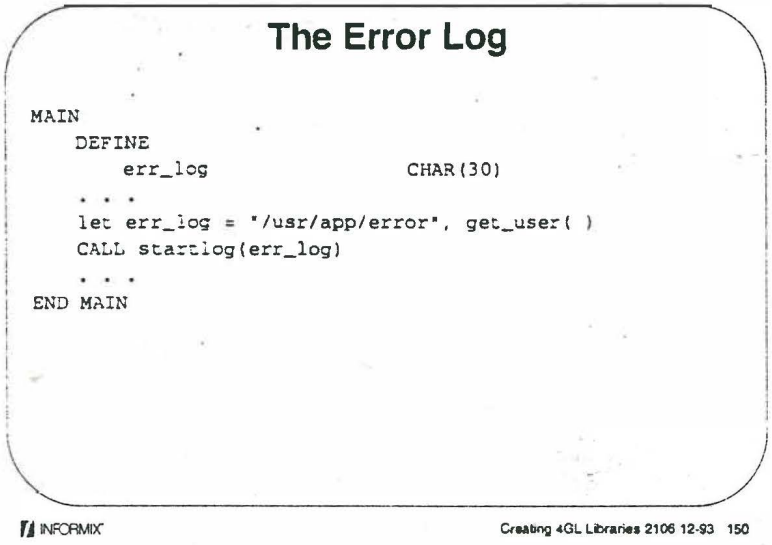
The WHENEVER ERROR CALL statement is useful for fatal error handling. You must be careful that the statement is only in effect for statements that should work but fail.

Fatal error processing functions should:

- Insert an entry in an error log file or database table.
- Provide as much information as possible to assist in the evaluation and debugging process.
- Display an error message to the user (if applicable).
- Terminate the program gracefully.

Warning!

If the *error_function* has a statement that fails, then the statement can cause recursive calls to the *error_function*. To avoid this, simply place a WHENEVER ERROR STOP statement at the beginning of your *error_function*.



The Error Log

4GL library functions provide the functionality to create and write to an error log file. The standard error logging functionality can limit the information related to the generation of the error. By implementing a few custom error handling functions to compliment the standard 4GL library functions, you can provide more information to assist the evaluation and debugging process.

startlog( )

The startlog(*" filename"*) library function creates an error file in which you can keep a record of run-time errors. The *"filename"* is the pathname of the error log file or a CHAR variable that evaluates to the name of the error log file. The file name can be specified for a particular user by calling a function that would return the user name and assigning the path to a CHAR variable string.

errorlog( )

The **errorlog(*str*)** library function writes a string constant or a CHAR variable in the current error log file along with:

- The current system date and time.
- The 4GL error message corresponding to the SQLCA.SQLCODE value.

The get_user( ) Function

```
FUNCTION get_user( )
    DEFINE
        user_id                 CHAR(18)

    SELECT USER
      INTO user_id
      FROM systables
     WHERE systables.tabid = 1

    RETURN user_id CLIPPED

END FUNCTION
```

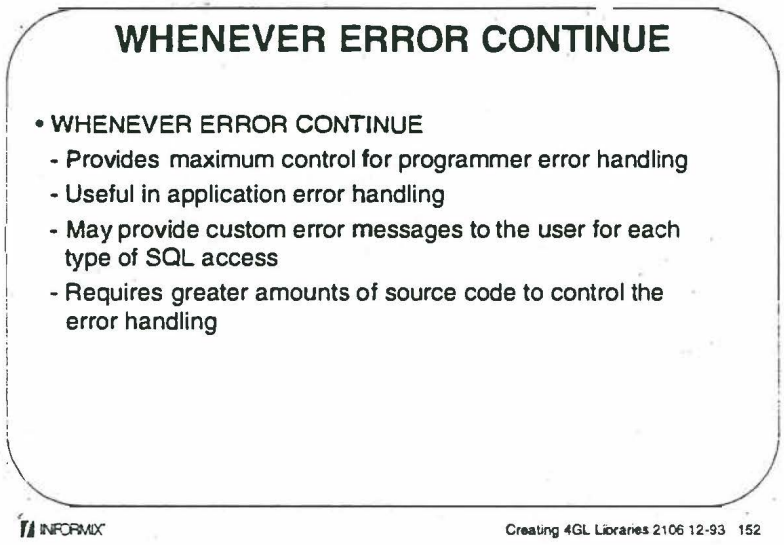

The get_user ( ) Function

Whether you choose to create error logs for each user or maintain a single error log shared by all users, recording the user id is a very important piece of error log information.

The get_user( ) function can be written several ways. C library function could obtain the information from the operating system, or a 4GL function could select the user. The statement must produce a singleton select to avoid the use of a cursor.

Tip: The SELECT statement will update the error codes in the SQL Communications Area. Therefore, it is important to save the error codes you will record in the error log prior to calling the get_user( ) function.

You should record as much information as possible in the error log to assist in the evaluation and debugging process.


WHENEVER ERROR CONTINUE
• WHENEVER ERROR CONTINUE
- Provides maximum control for programmer error handling
- Useful in application error handling
- May provide custom error messages to the user for each type of SQL access
- Requires greater amounts of source code to control the error handling
INFORMIX
Creating 4GL Libraries 2106 12-93 152

WHENEVER ERROR CONTINUE

The WHENEVER ERROR CONTINUE statement provides maximum control to the programmer for error handling. Therefore, it is very effective for application error handling.

Custom Error Messages

The power of WHENEVER ERROR CONTINUE is the ability to provide custom error messages to the user for each type of SQL access. The string returned from the 4GL **err_get( )** function is usually not acceptable to display to the end user. Custom error messages can be implemented by:

- hard coding the message at the specific point it is displayed to the user. You can display a literal string or declare a CASE statement to handle multiple error codes.
- storing custom error messages in a database table. You can assign a symbolic name or number to each message to facilitate the SELECT statement.

Disadvantages

The disadvantage of hard coding error messages is the code modification, re-compilation and version control required to add or change messages.

The disadvantage of using a database table is the extra database access required for every message.

Custom Error Messages

You can usually anticipate specific application errors that may be encountered by end users within a particular function or SQL access operation.

These errors should be handled to allow the end user to recover from the particular operation and not result in program termination.

An error message should be displayed to the end user informing them of the error (in terms they can understand) and instructing the user if actions are required.

Custom Error Messages

There are many ways to implement custom error messages. Two examples are shown below:

Hard Coded Messages

Example:

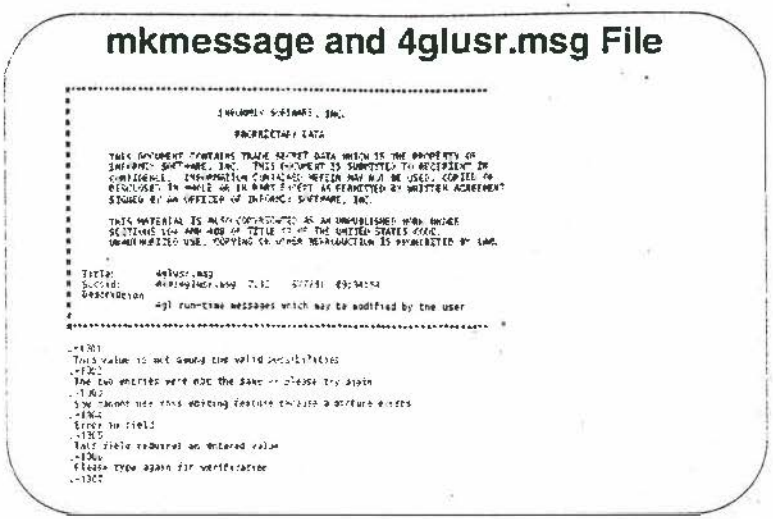
```
CASE
    WHEN (sql_stat = -239)
        ERROR "This Customer Already Exists"
    WHEN (sql_stat = -263)
        ERROR "The order is locked by another user, please try
              again"
. . .
END CASE
```

Database Table Messages

Example:

```
FUNCTION disp_err_msg(err_num)
    DEFINE
        err_num             INTEGER,
        err_msg             CHAR (70)

    SELECT message
        INTO  err_msg
        FROM  error_msg_tbl
        WHERE err_num = msg_num

    ERROR err_msg CLIPPED

END FUNCTION
```

mkmessage and 4glusr.msg File



mkmessage and 4glusr.msg File

You can use the **mkmessage** utility to customize 4GL run-time messages in the **4glusr.msg** file. INFORMIX-4GL supplies a run-time error message file called **4glusr.msg** located in the **$INFORMIXDIR/msg** directory.

The ASCII file contains run-time error messages that cannot be trapped with the WHENEVER ERROR statement and messages that support the 4GL **Help** menu.

The **4glusr.iem** is the executable version of the message file. If you modify the file, you must specify the source and object file name on the **mkmessage** command line:

```
mkmessage 4glusr.msg 4glusr.iem
```

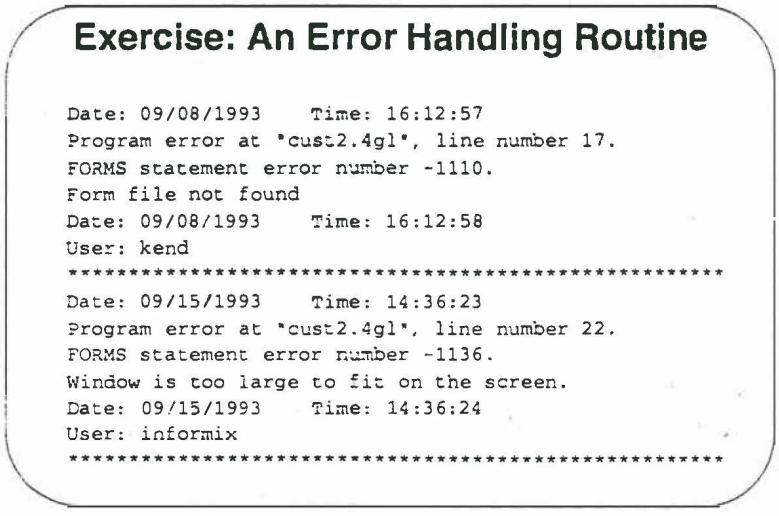
Warning! Do not alter the numeric error codes or the period (.) identifier. You can modify the error message text only.

A Library Makefile

```
.SUFFIXES:

.SUFFIXES: .o .ec .c .4gl .4go .4ga .4gi .per .frm .msg .iem

4GL = err_func.4gl get_user.4gl

FORM =

LIB_NAME = err_lib.a

COBJS = $(4GL:.4gl=.o)
CFORM = $(FORM:.per=.frm)

C_LIB:    $(COBJS)
          ar rc $(LIB_NAME) $(COBJS)
          @- cp $(LIB_NAME) $(BASEDIR)/src/4gl_lib
          @- echo "Finished compiling ccode..."

FORM:     $(CFORM)
          @- echo 'Finished compiling forms...'

.4gl.o:
               c4gl -c $*.4gl
               @- rm -f $*.c $*.ec
```

Replace PROG_NAME with LIB_NAME

Define a custom target called LIB that compiles the C objects (COBJS), archives the COBJS into LIB_NAME err_lib.a and copies the library archive file to the 4gl_lib directory.


Exercise: An Error Handling Routine
Date: 09/08/1993 Time: 16:12:57
Program error at "cust2.4gl", line number 17.
FORMS statement error number -1110.
Form file not found
Date: 09/08/1993 Time: 16:12:58
User: kend
****
Date: 09/15/1993 Time: 14:36:23
Program error at "cust2.4gl", line number 22.
FORMS statement error number -1136.
Window is too large to fit on the screen.
Date: 09/15/1993 Time: 14:36:24
User: informix
****
INFORMIX
Creating 4GL Libraries 2106 12-93 156

Exercise: An Error Handling Routine

Create a fatal error handling routine to be called by the WHENEVER ERROR CALL statement. The error routine should log the date, time, 4GL error message corresponding to the SQLCA.SQLCODE value and the user name. You should be in your **$BASEDIR/src/exercise2** directory.

Step 1

Create an error log function to be called by the WHENEVER ERROR CALL statement. Name this function **err_func( )**.

Step 2

Create a function to SELECT the user id and return it to the **err_func( )** function. Name this function **get_user( )**.

Note:

Create the **err_func( )** and **get_user( )** functions in separate **.4gl** files.

Step 3

Copy the **makefile** on the previous page into your **exercise2** directory. Run the **C_LIB** target to compile your application and create an archive library file in your **4gl_lib** directory.

Step 4

Change directories to **exercise1**.

Edit your stub file and call the **startlog** function. Name your error log **err_log**.

Edit your **cust_ord.4gl** module of your customer order application to include the WHENEVER ERROR CALL statement as the first statement in the first function.

Edit each function in the **svc_cust_ord.4gl** module of your customer order application to include the WHENEVER ERROR CALL statement. You want the statement to be in effect anywhere that you are not doing your own error checking. You want to *turn off* the WHENEVER ERROR CALL statement for any SQL access statements where you perform yo own error checking.

Step 5

Edit your **makefile**:

- Define the name of your 4GL library, **err_lib.a**. Be sure to include the path of your 4GL library directory.
- Include the library file on the compile line for the CCODE target.

Step 6

Run the CCODE target to re-compile your application and link in your 4GL library file.

Step 7

Simulate a fatal error to test your error handling routine. A fatal error can be simulated by:

- Removing (or renaming) your form specification file (**disp_win.frm**).
- Opening a window too small for the form screen to be displayed in and running your program.

Step 8

Examine the error log.

Bonus Exercise

Create the p-code library for the INFORMIX-4GL Rapid Development System at the command line.

Solutions

error.4gl

err_func( )

```
FUNCTION err_func ( )

    DEFINE

        err_log CHAR(255)

    WHENEVER ERROR STOP

    CLEAR SCREEN

    ERROR "Fatal Error: Program exiting . . . "

    LET err_log = "User: ", get_user ( ), "\n",
     "*********************************************************"

    CALL errorlog ( err_log CLIPPED )

    EXIT PROGRAM

END FUNCTION
```

get_user.4gl

get_user( )

```
FUNCTION get_user ( )

    DEFINE

        user_id CHAR(18)

    WHENEVER ERROR CONTINUE

    SELECT USER
      INTO user_id
      FROM systables
     WHERE systables.tabid = 1

    RETURN user_id CLIPPED

END FUNCTION
```

The Application's Makefile

```
.SUFFIXES:

.SUFFIXES: .o .ec .c .4gl .4go .4ga .4gi .per .frm .msg .iem

LIB_FILE = $(BASEDIR)/src/4gl_lib/err_lib.a
4GL = cust_1.4gl cust_2.4gl globals.4gl stub.4gl
FORM = disp_win.per

PROG_NAME = final_prog.4ge

COBJS = $(4GL:.4gl=.o)
POBJS = $(4GL:.4gl=.4go)
CFORM = $(FORM:.per=.frm)
RDS_NAME= $(PROG_NAME:.4ge=.4gi)

CCODE:   $(COBJS)
                    c4gl -o $(PROG_NAME) $(COBJS) $(LIB_FILE)
                    @- echo "Finished compiling ccode..."
PCODE:   $(POBJS)
                    @- cat $(POBJS) > $(RDS_NAME)
                    @- echo "Finished compiling pcode..."
FORM:   $(CFORM)
                    @- echo "Finished compiling forms..."

.4gl.4go:
                    fglpc $*.4gl
.4gl.o:
                    c4gl -c $*.4gl
                    @- rm -f $*.c $*.ec
.per.frm:
                    form4gl -s $*.per
```

stub.4gl

MAIN

```
GLOBALS "globals.4gl"
MAIN

    DEFINE
    a_customer_num LIKE customer.customer_num,
    sql_stat    INTEGER

    IF ( NUM_ARGS ( ) != 1 ) THEN

        DISPLAY "USAGE: main.4gs <customer number>"
        DISPLAY ""

    ELSE
        LET a_customer_num = ARG_VAL(1)

        CALL startlog ( "err_log" )

        CALL sel_ord_item_pre ( )
            RETURNING sql_stat

        IF ( NOT sql_stat ) THEN
            CALL cust_ctrl ( a_customer_num )
        ELSE
            DISPLAY "ERROR PREPARE/DECLARE ON items TABLE ",
            sql_stat
        END IF

    END IF

END MAIN
```

cust_ord.4gl

cust_ctrl( )

```
GLOBALS "globals.4gl"

FUNCTION cust_ctrl ( p_customer_num )

    DEFINE
        cust_rec     RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

    p_customer_num LIKE customer.customer_num,

    sql_stat INTEGER

    # WHENEVER ERROR STOP changed to
    WHENEVER ERROR CALL err_func

    OPEN WINDOW cust_win AT 2,3 WITH FORM "disp_win"
    ATTRIBUTE ( BORDER )

    CALL sel_cust_rec ( p_customer_num )
        RETURNING sql_stat, cust_rec.*

    IF ( sql_stat = NOTFOUND ) THEN
        ERROR "No customers found...."
        SLEEP 3
    ELSE
        CALL disp_cust ( cust_rec.* )

        CALL disp_ord_ctrl ( cust_rec.customer_num )

    END IF

    CLOSE WINDOW cust_win

END FUNCTION
```

disp_cust( )

```
FUNCTION disp_cust ( p_cust_rec )

    DEFINE

        p_cust_rec   RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

    answer CHAR(1)

    DISPLAY BY NAME p_cust_rec.*

END FUNCTION
```

disp_ord_ctrl( )

```
FUNCTION disp_ord_ctrl ( p_customer_num )

    DEFINE

        p_customer_num LIKE customer.customer_num,
        sql_stat INTEGER,
        row_cnt  SMALLINT,
        answer   CHAR(1)

    CALL sel_cust_ord ( p_customer_num )
        RETURNING sql_stat, row_cnt

    IF ( row_cnt = 0 ) THEN
        ERROR "Customer has NO orders"
        PROMPT "Press any key to continue...." FOR CHAR answer
    ELSE

    CALL disp_ord ( row_cnt )

    END IF

END FUNCTION
```

disp_ord()

```
FUNCTION disp_ord ( row_cnt )

    DEFINE

        sql_stat      INTEGER,
        item_row_cnt  SMALLINT,
        row_cnt       SMALLINT,
        cont_inp      SMALLINT,
        cur_row       SMALLINT,
        scr_row       SMALLINT

    CALL SET_COUNT ( row_cnt )

    LET cont_inp = TRUE

    WHILE ( cont_inp )

        INPUT ARRAY ga_ord_rec WITHOUT DEFAULTS FROM sa_ord_rec.*

            AFTER FIELD trap

                LET ga_ord_rec[cur_row].trap = NULL
                DISPLAY "" TO sa_ord_rec[scr_row].trap

            AFTER ROW

                LET cur_row = ARR_CURR ( )
                LET scr_row = SCR_LINE ( )

                DISPLAY ga_ord_rec[cur_row].* TO
                        sa_ord_rec[scr_row].*

            BEFORE ROW

                LET cur_row = ARR_CURR ( )
                IF ( cur_row = row_cnt + 1 ) THEN
                    LET cont_inp = TRUE
                    EXIT INPUT
                ELSE
                    LET scr_row = SCR_LINE ( )
                    DISPLAY ga_ord_rec[cur_row].* TO
                            sa_ord_rec[scr_row].*
                        ATTRIBUTE ( REVERSE )
                    LET cont_inp = FALSE
```

```
                CALL proc_items ( cur_row )
                    RETURNING sql_stat, item_row_cnt
            END IF

        ON KEY ( CONTROL-V )
            CALL disp_array_items ( item_row_cnt )

    END INPUT

END WHILE

END FUNCTION
```

proc_items( )

```
FUNCTION proc_items ( p_cur_row )

    DEFINE
        sql_stat INTEGER,

        p_cur_row SMALLINT,
        item_row_cnt SMALLINT

    CALL sel_ord_item ( p_cur_row
        RETURNING sql_stat, item_row_cnt

    CALL disp_four_items ( )

    RETURN sql_stat, item_row_cnt

END FUNCTION
```

disp_four_items( )

```
FUNCTION disp_four_items ( )

    DEFINE
        i SMALLINT

    FOR i = 1 TO 4
        DISPLAY ga_item_ord[i].* TO sa_item[i].*
    END FOR

END FUNCTION
```

disp_array_items( )

```
FUNCTION disp_array_items   p_item_row_cnt

    DEFINE
        p_item_row_cnt SMALLINT

    CALL SET_COUNT ( p_item_row_cnt

    DISPLAY ARRAY ga_item_ord TO sa_item.*

        ON KEY ( CONTROL-V
            EXIT DISPLAY

    END DISPLAY
END FUNCTION
```

svc_cust_ord.4gl

sel_cust_rec( )

```
GLOBALS "globals.4gl"

FUNCTION sel_cust_rec ( p_customer_num )

    DEFINE

        cust_rec    RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            company      LIKE customer.company
        END RECORD,

        p_customer_num LIKE customer.customer_num,

        sql_stat INTEGER

    WHENEVER ERROR CONTINUE

    SELECT customer.customer_num, customer.fname,
           customer.lname, customer.company
    INTO cust_rec.*
    FROM customer
    WHERE customer.customer_num = p_customer_num

    # WHENEVER ERROR STOP changed to
    WHENEVER ERROR CALL err_func

    LET sql_stat = SQLCA.SQLCODE

    RETURN sql_stat, cust_rec.*

END FUNCTION
```

sel_cust_ord( )

```
FUNCTION sel_cust_ord ( p_customer_num )

    DEFINE
        p_customer_num  LIKE customer.customer_num,
        sql_stat        INTEGER,
        row_cnt         SMALLINT

    WHENEVER ERROR CONTINUE

    DECLARE sel_cust_ord_curs CURSOR FOR
     SELECT * , orders.order_num, orders.order_date,
            orders.po_num, orders.ship_date,
            orders.ship_charge, orders.paid_date
       FROM orders
      WHERE orders.customer_num = p_customer_num

    # WHENEVER ERROR STOP changed to
    WHENEVER ERROR CALL err_func

    LET sql_stat = SQLCA.SQLCODE

    LET row_cnt = 1

    IF ( NOT sql_stat ) THEN

        WHENEVER ERROR CONTINUE

        FOREACH sel_cust_ord_curs INTO ga_ord_rec[row_cnt].*
            # WHENEVER ERROR STOP changed to
            WHENEVER ERROR CALL err_func
             1
            LET sql_stat = SQLCA.SQLCODE

            LET row_cnt = row_cnt + 1

            IF ( row_cnt > 75 ) THEN
                EXIT FOREACH
            END IF

        END FOREACH
    END IF

    RETURN sql_stat, row_cnt - 1
END FUNCTION
```

sel_ord_item_pre()

```
FUNCTION sel_ord_item_pre ( )

    DEFINE

        sql_text CHAR(250),
        sql_stat INTEGER

    WHENEVER ERROR CALL err_func

    LET sql_text = " SELECT items.item_num, items.stock_num, ",
                         " items.manu_code, items.quantity, "
                         " items.total_price "
                    " FROM items ",
                   " WHERE items.order_num = ? "

    WHENEVER ERROR CONTINUE

    PREPARE sel_item_stmt FROM sql_text

    # WHENEVER ERROR STOP changed to
    WHENEVER ERROR CALL err_func

    LET sql_stat = SQLCA.SQLCODE

    IF ( NOT sql_stat ) THEN

        WHENEVER ERROR CONTINUE

        DECLARE sel_item_curs CURSOR FOR sel_item_stmt

        # WHENEVER ERROR STOP changed to
        WHENEVER ERROR CALL err_func

         LET sql_stat = SQLCA.SQLCODE

    END IF

    RETURN sql_stat

END FUNCTION
```

sel_ord_item()

```
FUNCTION sel_ord_item ( p_cur_row )

    DEFINE
        sql_stat INTEGER,

        p_cur_row SMALLINT,
        row_cnt SMALLINT

    WHENEVER ERROR CALL err_func

    CALL null_items ( )

    WHENEVER ERROR CONTINUE

    OPEN sel_item_curs USING ga_ord_rec[p_cur_row].order_num

    # WHENEVER ERROR STOP changed to
    WHENEVER ERROR CALL err_func

    LET sql_stat = SQLCA.SQLCODE
    LET row_cnt = 1

    WHILE ( ( NOT sql_stat ) AND ( row_cnt <= 100 ) )

        WHENEVER ERROR CONTINUE
        FETCH sel_item_curs INTO ga_item_ord[row_cnt].*

        # WHENEVER ERROR STOP changed to
        WHENEVER ERROR CALL err_func

        LET sql_stat = SQLCA.SQLCODE

        IF ( NOT sql_stat ) THEN
            LET row_cnt = row_cnt + 1
        END IF

    END WHILE

    IF ( sql_stat = 100 ) THEN
        LET sql_stat = 0
    END IF

    RETURN sql_stat, row_cnt - 1
END FUNCTION
```

null_items()

```
FUNCTION null_items ( )

    DEFINE
        i SMALLINT

    WHENEVER ERROR CALL err_func

    INITIALIZE ga_item_ord[1].* TO NULL

    FOR i = 2 TO 100
        LET ga_item_ord[i].* = ga_item_ord[1].*
    END FOR

END FUNCTION
```

Chapter

8

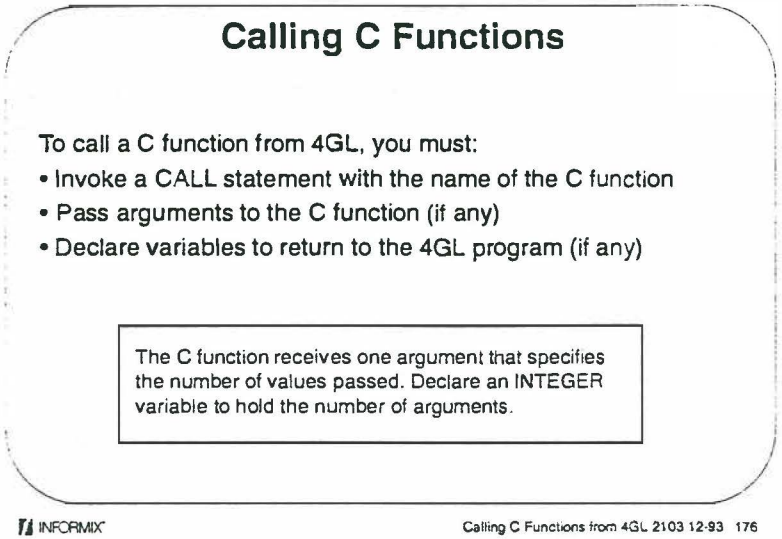
Calling C Functions from 4GL

Objectives

At the end of this module, you will be able to:

- Explain the use of the 4GL argument stack
- Describe pop, push and return library functions
- Use library functions to manipulate the argument stack
- Call a C function from a 4GL program
- Explain how to call a 4GL function from a C program

© 1993 Informix Software, Inc


Calling C Functions
To call a C function from 4GL, you must:
• Invoke a CALL statement with the name of the C function
• Pass arguments to the C function (if any)
• Declare variables to return to the 4GL program (if any)
The C function receives one argument that specifies the number of values passed. Declare an INTEGER variable to hold the number of arguments.
INFORMIX
Calling C Functions from 4GL 2103 12-93 176

Calling C Functions

The C function receives one formal integer argument that specifies the number of values pushed on the argument stack. Declare an INTEGER variable to hold this value. The C function acquires the actual values by popping the arguments off the stack.

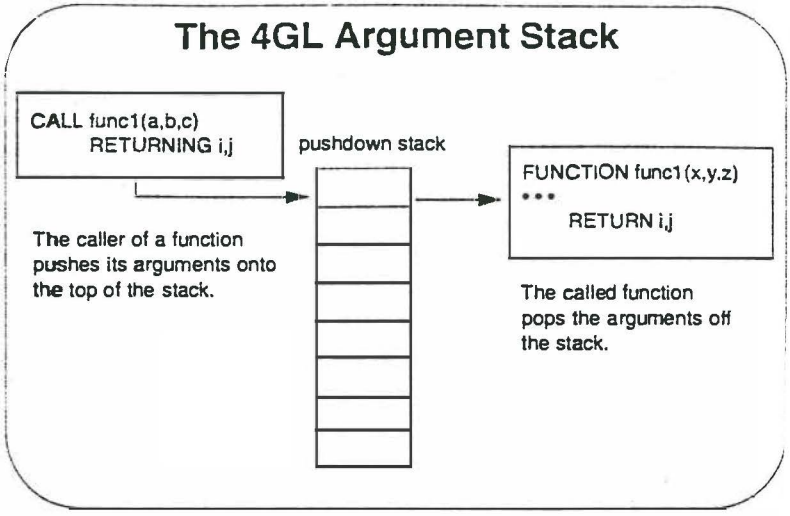
Handling Arguments

When using the INFORMIX-4GL Rapid Development System and calling C functions, the maximum number of arguments it permits is defined in the **fglusr.c** file. The 4GL function cannot call it with more arguments. However, the INFORMIX-4GL C Compiler Version has no information about the external function. It will permit a 4GL function to call a C function with any number of arguments.

Therefore, you should test for the correct number of arguments pushed on the stack by the calling function. If an incorrect number of arguments are passed to the C function, display a usage statement.

Example

```
int c_function ( args )
int args ;
. . .
if ( args != 2 )
{
    fprintf ("c_function USAGE: <arg1> <arg2> )\n" ) ;
    return (2) ;
. . .
}
```


The 4GL Argument Stack
CALL func1(a,b,c)
RETURNING i,j
pushdown stack
FUNCTION func1(x,y,z)
...
RETURN i,j
The caller of a function pushes its arguments onto the top of the stack.
The called function pops the arguments off the stack.
INFORMIX
Calling C Functions from 4GL 2103 12-93 177

The 4GL Argument Stack

4GL uses a pushdown argument stack to pass values between 4GL functions. When a 4GL program executes a CALL statement, 4GL first notes the current argument stack depth.

Push and Pop

The caller of the function pushes its arguments onto the top of the stack from left to right. A *push* places *two* elements onto the stack: the value of the argument and its data type.

The called function pops the arguments off the stack into local variables If the type declaration of the variables is different from the argument types, 4GL can convert arguments to the proper type.

The return statement pushes the returned values onto the stack from left to right. The RETURNING clause in the originating CALL statement pops the values off the stack.

The 4GL Argument Stack and C functions

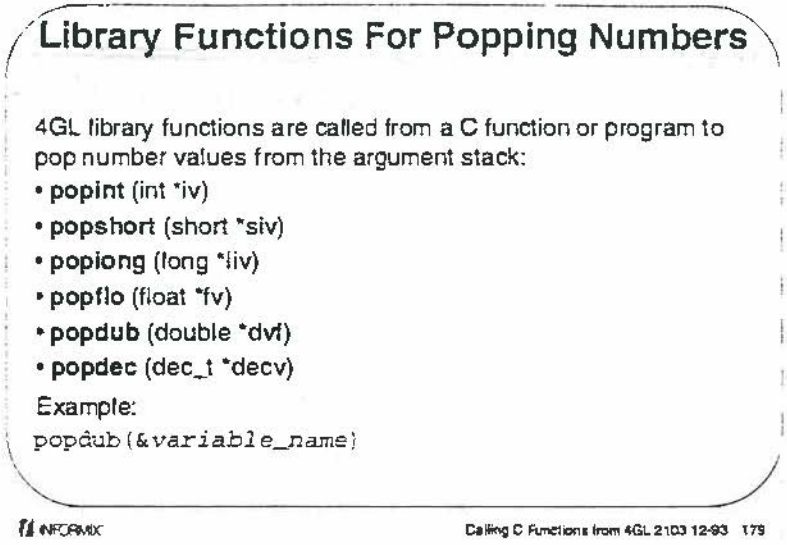
The 4GL argument stack is also used when a 4GL program calls a C function with one primary difference. You push and pop values using external push and pop library functions.

Equivalent C and SQL Data Types

SQL Data Type	C Data Type	Pop Function	Push Function
integer	int i	popint(&i)	retint(i)
smallint	short s	popshort(&s)	retshort(s)
integer	long l	poplong(&l)	retlong(l)
smallfloat	float f	popflo(&f)	retflo(&f)
float	double d	popdub(&d)	retdub(&d)
*char(m-1)	char str[m]	popquote(str,m)	retquote(str)
decimal,money	dec_t dm	popdec(&dm)	retdec(&dm)

Equivalent C and SQL Data Types

The SQL data types and equivalent C data types are shown above. There are two built-in C library functions for each data type. The C library functions are covered in detail in the following pages.

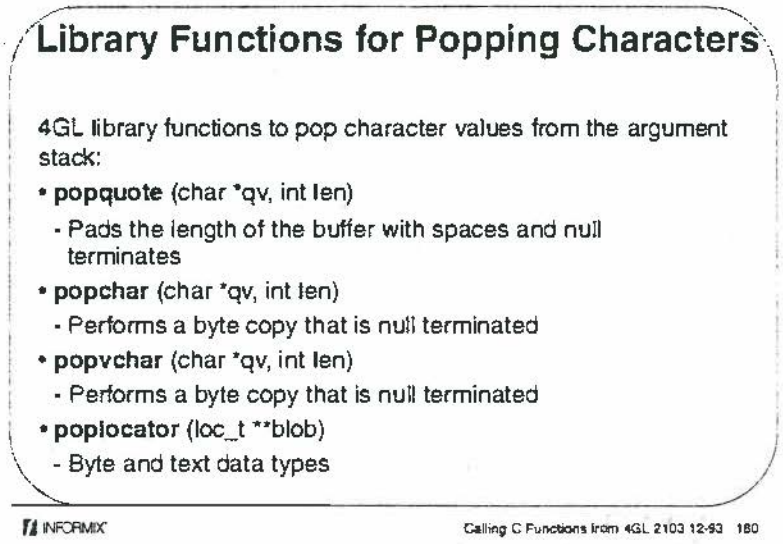

Library Functions For Popping Numbers
4GL library functions are called from a C function or program to pop number values from the argument stack:
• popint (int *iv)
• popshort (short *siv)
• poplong (long *liv)
• popflo (float *fv)
• popdub (double *dvf)
• popdec (dec_t *decv)
Example:
popdub(&variable_name)
INFORMIX
Calling C Functions from 4GL 2.03 12-93 179

Library Functions for Popping Numbers

A C function receives variables from 4GL by using the argument stack. The C function must use pop external library functions provided with INFORMIX-4GL. All of the external pop functions perform the following steps:

1. Removes one value from the top of the stack
2. Converts its data type if necessary. If the value on the stack cannot be converted to the specified data type, the result is undefined.
3. Copies it to the designated variable

For more information about external pop library functions, refer to the INFORMIX-ESQL/C *Version 4.0 Programmer Manual* and the INFORMIX-4GL *Reference Manual*.


Library Functions for Popping Characters
4GL library functions to pop character values from the argument stack:
• popquote (char *qv, int len)
- Pads the length of the buffer with spaces and null terminates
• popchar (char *qv, int len)
- Performs a byte copy that is null terminated
• popvchar (char *qv, int len)
- Performs a byte copy that is null terminated
• poplocator (loc_t **blob)
- Byte and text data types
INFORMIX
Calling C Functions from 4GL 2103 12-93 180

Library Functions for Popping Characters

The **popquote**, **popchar** and **popvchar** functions copy exactly **len** bytes into the string buffer ***qv**. The **len** argument sets the maximum size of the string buffer.

Note an important difference between the functions. The **popquote** function pads the length of the byte string buffer with spaces. The **popchar** and **popvchar** function final byte copied to the buffer is null terminated. Therefore, the maximum string length is **len-1**. If the stack argument is longer than **len-1**, the trailing bytes are lost.

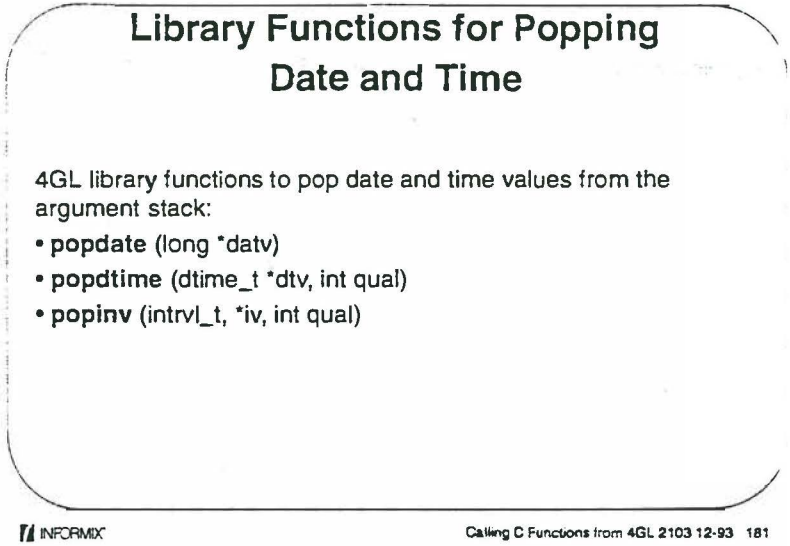
Popquote

You will always receive exactly **len** bytes (trailing spaces and a null value) when using **popquote**. You must trim the trailing spaces from the value. Even if the value on the stack is an empty string, you will receive exactly **len** bytes.

Poplocator

The **poplocator** function is used when popping BYTE or TEXT data type with INFORMIX-OnLine. The **poplocator** does not copy the value into the designated variable, only its address.

For more information about the poplocator library function, refer to the INFORMIX-OnLine *Programmer Manual*.


Library Functions for Popping Date and Time
4GL library functions to pop date and time values from the argument stack:
• popdate (long *datv)
• popdtime (dtime_t *dtv, int qual)
• popinv (intrvl_t, *iv, int qual)
INFORMIX
Calling C Functions from 4GL 2103 12-93 181

Library Functions for Popping Date and Time

The functions **popdate**, **popdtime** and **popinv** are used to pop DATE, DATETIME and INTERVAL values. The **qual** argument receives the binary representation of DATETIME or INTERVAL.

For more information about the date and time library function, refer to the INFORMIX-ESQL/C *Version 4.0 Programmer Manual*.

Return and Push Library Functions

A C function can pass one or more arguments to 4GL by placing the arguments on the stack using:

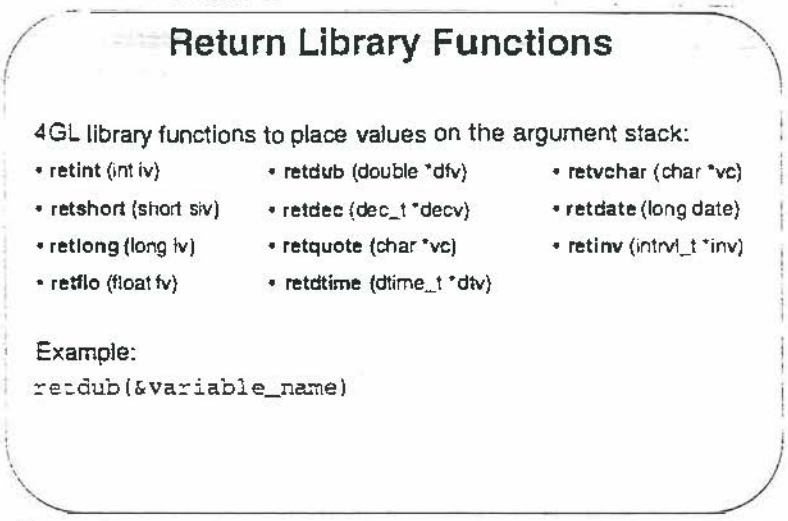
- **Return** library functions
  - The **return** functions make an internal copy of the arguments and place the arguments on the stack. Although the storage allocated for the argument is released when the function ends, the stack argument still exists.
- **Push** library functions
  - The **push** library functions do not make a copy of the arguments and place only a pointer to the argument on the stack. The arguments must be valid for the duration of the call.

Return and Push Library Functions

If you are passing values to a 4GL program from a C function, use the return library functions. The return functions make an internal copy of the arguments and place the copy of the arguments on the stack. Even if the storage is de-allocated for the internal copy, you have access to the argument stack. When you are returning values from local variables, you should always use the return library functions.

If you are passing values to a 4GL function from a C program, you may use the push library functions. The push functions only place a pointer to the argument on the stack. The pushed arguments must be valid for the duration of the call (a global variable), and it is up to the C code to release the values after the call.

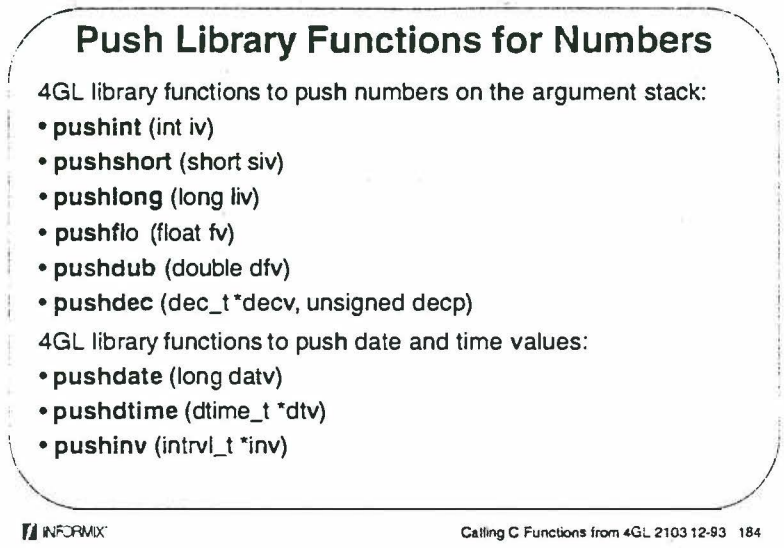
Warning! Only use push functions if you are familiar with memory management and allocation in C programming.


Return Library Functions
4GL library functions to place values on the argument stack:
• retint (int iv)
• retdub (double *dfv)
• retvchar (char *vc)
• retshort (short siv)
• retdec (dec_t *decv)
• retdate (long date)
• retlong (long lv)
• retquote (char *vc)
• retinv (intrvl_t *inv)
• retflo (float fv)
• retdtime (dtime_t *dtv)
Example:
retdub(&variable_name)
INFORMIX
Calling C Functions from 4GL 2103 12-93 183

Return Library Functions

The 4GL library functions available to place values on the stack are listed above. The C function can return any data type. 4GL will convert the data type as required when it pops the value off of the stack. If the conversion fails, an error will be returned.

Note: A C function called from 4GL must always exit the function with the statement return (n), where n is the number of return values placed on the stack. If the functions do not return a value, then the statement must return (0).


Push Library Functions for Numbers
4GL library functions to push numbers on the argument stack:
• pushint (int iv)
• pushshort (short siv)
• pushlong (long liv)
• pushflo (float fv)
• pushdub (double dfv)
• pushdec (dec_t *decv, unsigned decp)
4GL library functions to push date and time values:
• pushdate (long datv)
• pushdtime (dtime_t *dtv)
• pushinv (intrvl_t *inv)
INFORMIX
Calling C Functions from 4GL 2103 12-93 184

Push Library Functions for Numbers

The 4GL library functions available to push number, date and time values on the stack are listed above.

The **dec_t** structure type in the **pushdec** function and C functions for manipulating decimal data are discussed in the INFORMIX-ESQL/C manual. The **decp** argument specifies the decimal precision and scale. To give a decimal variable the precision of 8 and the scale of 2, you would specify the following:

```
pushdec (dec_variable, PRECMAKE(8,2));
```

The PRECMAKE macro is defined in the **decimal.h** file.

Push Library Functions for Characters

4GL library functions to push character values on the argument stack:

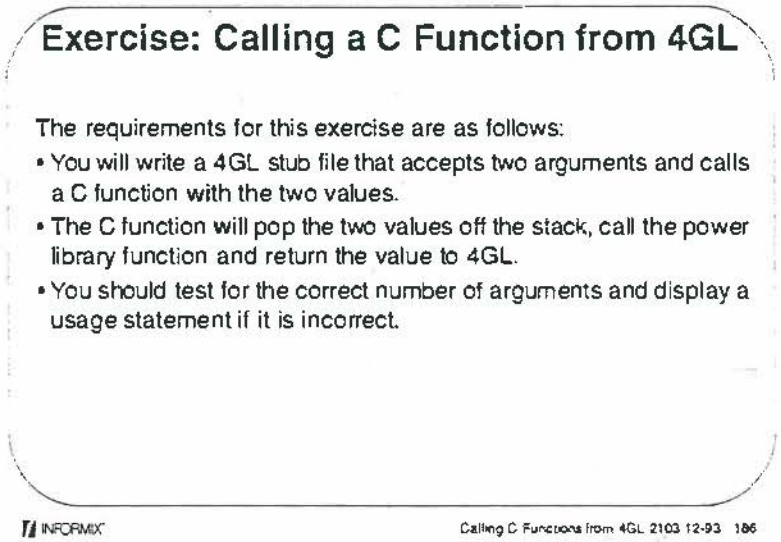
- **pushquote** (char *cv, int len)
- **pushvchar** (char *vcv, int len)
- **pushlocator** (loc_t *blob)
  - Byte and text data types

Push Library Functions for Characters

The arguments to **pushquote** and **pushvchar** functions are an unterminated character string and the length of characters that it contains.

Pushlocator

The **pushlocator** function is used when pushing BYTE or TEXT data types with INFORMIX-OnLine. For more information about the **pushlocator** library function, refer to the INFORMIX-OnLine *Programmer Manual*.


Exercise: Calling a C Function from 4GL
The requirements for this exercise are as follows:
• You will write a 4GL stub file that accepts two arguments and calls a C function with the two values.
• The C function will pop the two values off the stack, call the power library function and return the value to 4GL.
• You should test for the correct number of arguments and display a usage statement if it is incorrect.
INFORMIX
Calling C Functions from 4GL 2103 12-93 186

Exercise: Calling a C Function from 4GL

You should be in your **$BASEDIR/src/exercise3** directory to create your 4GL stub file and C function.

The **power** function is part of the *include* file in C, **math.h**. This file must be included in your C function. The format is **pow(x, y)**. It accepts two arguments of type *double* and returns a value of type *double*.

Step 1

Create a 4GL stub file that accepts two arguments of type *decimal*. It should display a usage statement if there is an incorrect number of arguments. It should then call a C function that passes the two arguments and returns the result of type *decimal*. Name the C function **pwr.c**.

Step 2

Copy and edit the C function provided to call the appropriate pop library function to pop the two values off the stack, call the power function, and call the appropriate return library function to return the result.

Step 3

Compile the 4GL file and C function at the command line using **c4gl**. Name the output executable file **power.4ge**. You need to include a **-lm** flag at the *end* of the **c4gl** line to include the mathematical library in C. You are performing the compile from the command line instead of using a **makefile** to facilitate the lab continued in the next module.

A sample **makefile** for defining and compiling C objects is located in the Appendix section of this manual.

wr.c ( )

```
#include <stdio.h>
#include <stdlib.h>
#include "math.h"

int pwr ( args )
int args;
{
    double p_number ;    /* the base number */
    double p_power ;     /* the exponent */
    double ret_val ;     /* the return value */

    ret_val = 0 ;        /* initialize the return value */
    .

    .
    ret_val = pow ( p_number, p_power ) ;
    .

    return ( 1 ) ;       /* indicates one argument is returned */

}
```

Solutions

stub.4gl

```
MAIN

    DEFINE

        arg_number  DECIMAL,
        arg_power   DECIMAL,
        ret_value   DECIMAL

    IF ( NUM_ARGS() != 2 ) THEN

        DISPLAY "USAGE: power.4ge <base number> <power>"

    ELSE

        LET arg_number = ARG_VAL(1)
        LET arg_power  = ARG_VAL(2)

        CALL pwr ( arg_number, arg_power )
            RETURNING ret_value

        DISPLAY ret_value

    END IF

END MAIN
```

pwr.c

```
#include <stdio.h>
#include <stdlib.h>
#include "math.h"

int pwr ( args )
int args;
{
    double p_number ;    /* the base number */
    double p_power ;     /* the exponent */
    double ret_val ;     /* the return value */

    ret_val = 0 ;        /* initialize the return value */

    popdub ( &p_power ) ;
    popdub ( &p_number ) ;

    ret_val = pow ( p_number, p_power ) ;

    retdub ( &ret_val ) ;

    return ( 1 ) ;      /* indicates one argument is returned */

}
```

c4gl Command Line

```
% c4gl -o power.4ge stub.4gl pwr.c -lm
```

Calling a 4GL Function from a C Program

INFORMIX-4GL provides the Application Programming Interface (API) to support 4GL function calls from a C program. To write a C program that calls a 4GL function, you must:

- Include the **fglapi.h** header file
- Execute the **fgl_start** macro to perform initialization tasks.
- Execute the **fgl_call** macro to call each 4GL function.
- Execute the **fgl_exitfm** macro if your 4GL function displays forms.
- Execute the fgl_end macro at the end of the program to free resources.

Calling a 4GL Function from a C Program

INFORMIX-4GL provides an API to support 4GL function calls from a C program. There are API functions for INFORMIX-4GL C Compiler Version and INFORMIX-4GL Rapid Development System. To pass values between the C program and 4GL function, use the push and pop functions described earlier in the module.

The fglapi.h File

The **fglapi.h** file defines the **fgl_start**, **fgl_call**, **fgl_exitfm** and **fgl_end** macros.

fgl_start

You must initialize the argument stack before you can call a 4GL function. The **fgl_start** macro will:

- Initialize the argument stack so you can pass arguments between C and 4GL.
- If you are using the RDS version, it specifies the file containing the 4GL functions.

fgl_call

You must use the **fgl_call** macro to execute a 4GL function. The **fgl_call** macro identifies the name of the 4GL function and the number of expected arguments.

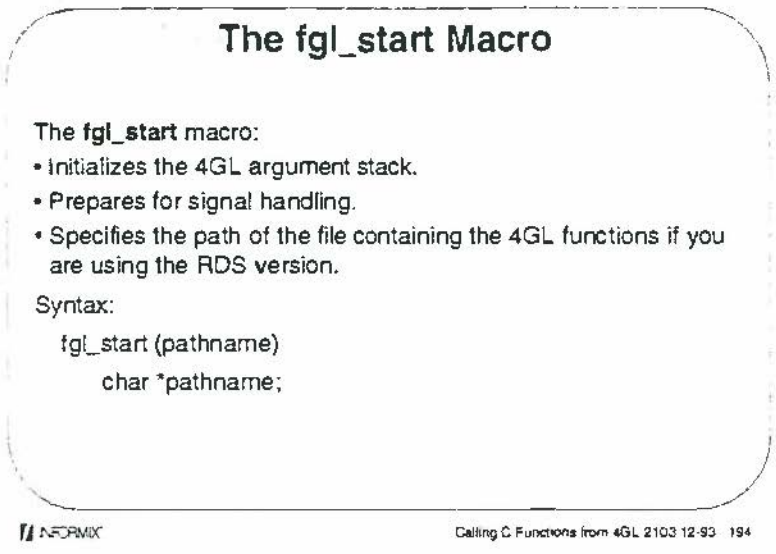
The fglapi.h Header File

You must include the **fglapi.h** header file in any C program that calls a 4GL function as follows:

```
#include <fglapi.h>
main( )
{
...
}
```


The fglapi.h Header File

The **fglapi.h** header file defines the **fgl_start**, **fgl_call**, **fgl_exitfm** and **fgl_end** macros. The file is located in the **SINFORMIXDIR/incl** directory. The header file must be included in any C program that calls a 4GL function.


The fgl_start Macro
The fgl_start macro:
• Initializes the 4GL argument stack.
• Prepares for signal handling.
• Specifies the path of the file containing the 4GL functions if you are using the RDS version.
Syntax:
fgl_start (pathname)
char *pathname;
INFORMIX
Calling C Functions from 4GL 2103 12-93 194

The fgl_start Macro

The **fgl_start** macro initializes the 4GL argument stack and prepares for signal handling. If you are using the RDS version, it specifies the path of the file containing the 4GL functions. The file extension, .4go or .4gi, is optional. This macro needs to be executed once for each C program.

You must call the **fgl_start** macro before:

- The **fgl_call** macro
- The 4GL push or pop function calls
- The **int_flag** or **quit_flag** globals integers are used for handling interrupts

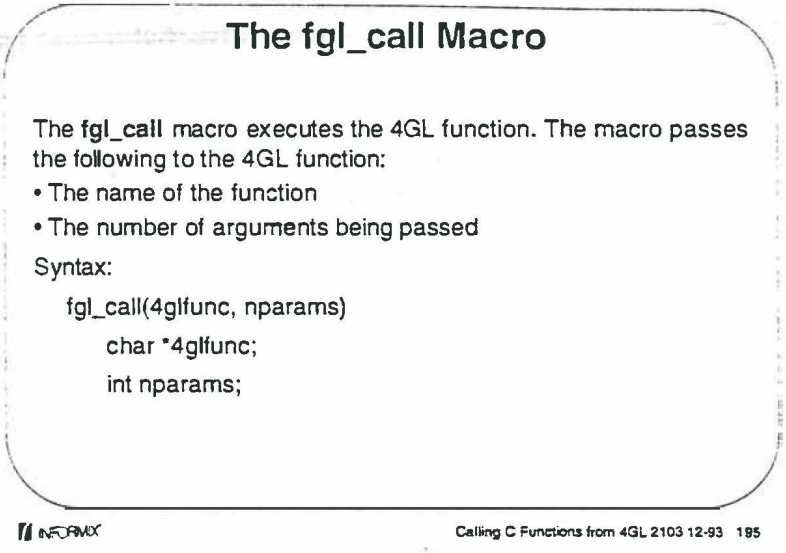
Example:

```
#include <fglapi.h>
main( )
{
fgl_start( );
. . .
}
```

Return Codes

The **fgl_start** macro returns the following status codes:

0	The macro executed successfully
<0	The macro failed


The fgl_call Macro
The fgl_call macro executes the 4GL function. The macro passes the following to the 4GL function:
• The name of the function
• The number of arguments being passed
Syntax:
fgl_call(4glfunc, nparams)
char *4glfunc;
int nparams;
INFORMIX
Calling C Functions from 4GL 2103 12-93 195

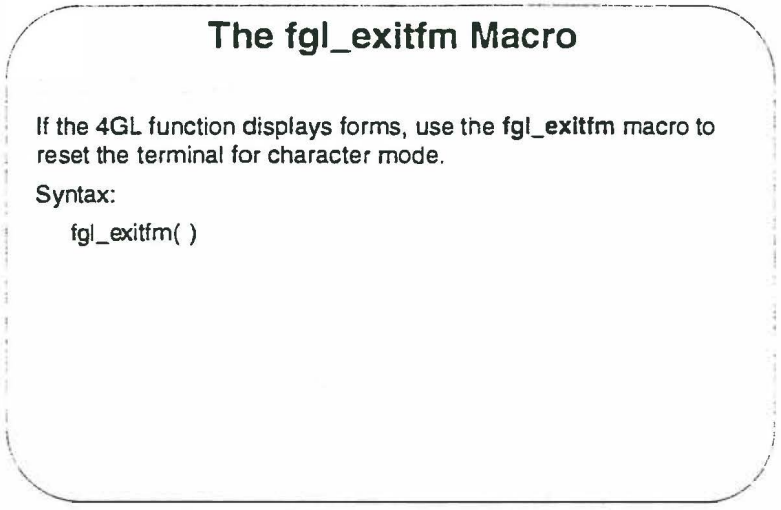
The fgl_call Macro

The fgl_call macro calls the 4GL function to execute. The macro passes the name of the function and the number of arguments. The macro returns the number of arguments being passed back to the program from the function.

You must push any values to be passed to the 4GL function onto the argument stack *before* executing the fgl_call macro.

Example

```
#include <fglapi.h>
main( )
{
. . .
    {
    fgl_start( );
    . . .
    pushquote ( arg_1, strlen (arg_1 ) );
    pushquote ( arg_2, strlen ( arg_2 ) );
    pushint ( arg_3 );
    fgl_call (4gl_func, 3);
    . . .
    }
. . .
}
```

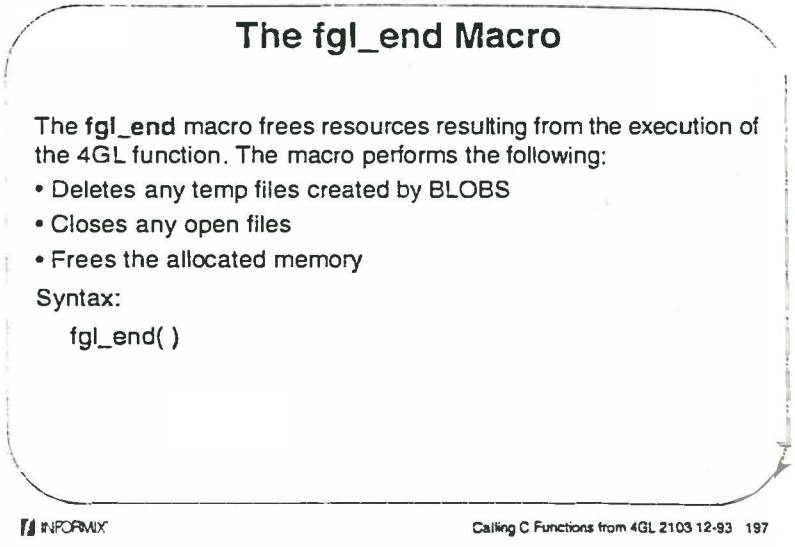

The fgl_exitfm Macro
If the 4GL function displays forms, use the fgl_exitfm macro to reset the terminal for character mode.
Syntax:
fgl_exitfm( )
INFORMIX
Calling C Functions from 4GL 2103 12-93 196

The fgl_exitfm Macro

The **fgl_exitfm** macro resets your terminal for character mode. Call the **fgl_exitfm** macro after any **fgl_call** macro that causes 4GL to display forms. If you do not execute the fgl_exitfm macro, your terminal may behave unusually after displaying a 4GL form.

Example

```
#include <fglapi.h>
main( )
{
. . .
    {
    fgl_start( );
    . . .
    pushquote (arg_1, strlen( arg_1 ));
    fgl_call (4gl_func, 1); /* 4gl_func displays one or more forms */
    popquote (arg_1, 75);
    fgl_exitfm( );
    . . .
    }
. . .
}
```

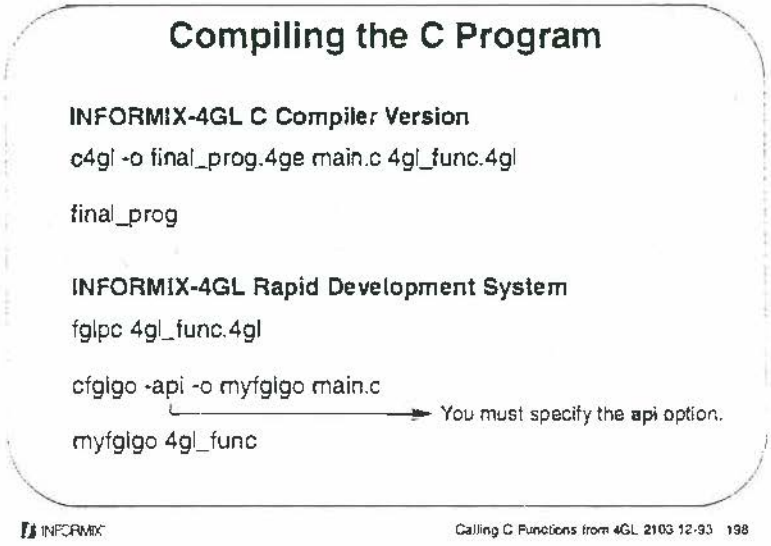

The fgl_end Macro
The fgl_end macro frees resources resulting from the execution of the 4GL function. The macro performs the following:
• Deletes any temp files created by BLOBS
• Closes any open files
• Frees the allocated memory
Syntax:
fgl_end( )
INFORMIX
Calling C Functions from 4GL 2103 12-93 197

The fgl_end Macro

The **fgl_end** macro frees resources associated with the execution of the 4GL function. The macro needs to be executed at the end of a C program that calls a 4GL function.

Example

```
#include <fglapi.h>
main( )
{
. . .
    {
    fgl_start( );
    . . .
    pushquote (arg_1, strlen( arg_1 ));
    fgl_call (4gl_func, 1); /* 4gl_func displays one or more forms */
    popquote (arg_1, 75);
    fgl_exitfm( );
    . . .
    fgl_end( );
    }
. . .
}
```



Compiling the C Program

The syntax and usage of the **c4gl** and **fglpc** commands are the same by which you compile and execute 4GL programs. The **cfglgo** command uses the **-api** option.

Note that you do not specify the **fgiusr.c** file on the command line of **cfglgo** unless you are calling external C functions from 4GL.

INFORMIX

Chapter

9

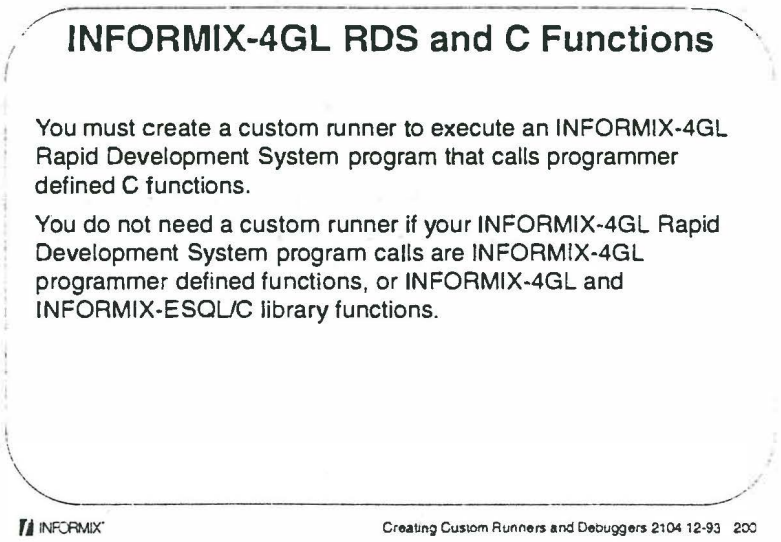
Creating Custom Runners and Debuggers

Objectives

At the end of this module, you will be able to:

- Create a custom runner and debugger
- Create a **fgiusr.c** file for calling C functions
- Use the **cfglgo** command
- Use the **cfgldb** command
- Understand the different implementations of the **fgiusr.c** file

© 1993 Informix Software, Inc.

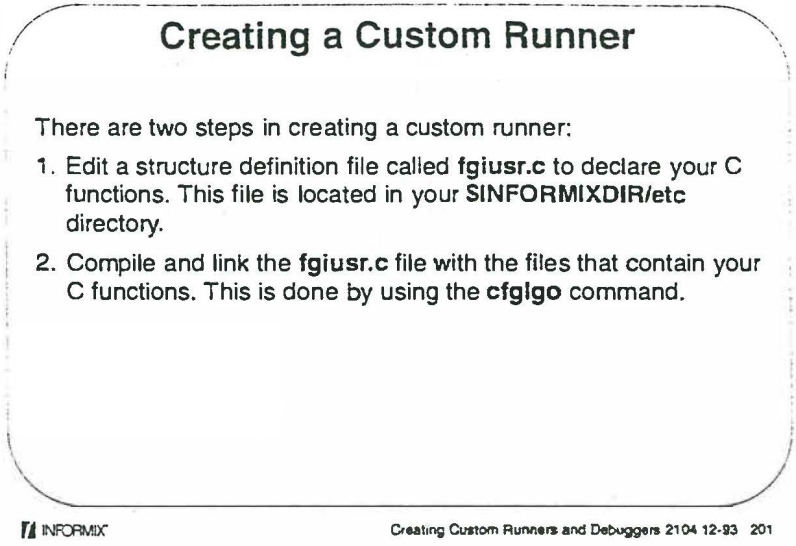

INFORMIX-4GL RDS and C Functions
You must create a custom runner to execute an INFORMIX-4GL Rapid Development System program that calls programmer defined C functions.
You do not need a custom runner if your INFORMIX-4GL Rapid Development System program calls are INFORMIX-4GL programmer defined functions, or INFORMIX-4GL and INFORMIX-ESQL/C library functions.
INFORMIX®
Creating Custom Runners and Debuggers 2104 12-93 200

INFORMIX-4GL RDS and C Functions

INFORMIX-4GL is a very powerful fourth-generation database programming language based on industry standard SQL. It can be used to develop complex programs that meet most application needs. There are special application requirements where the C programming language can be used to enhance or complement the functionality of INFORMIX-4GL.

No fglgo

If your INFORMIX-4GL Rapid Development System program calls programmer defined C functions, you must create a custom runner and debugger. You cannot use the default p-code runner **fglgo**.

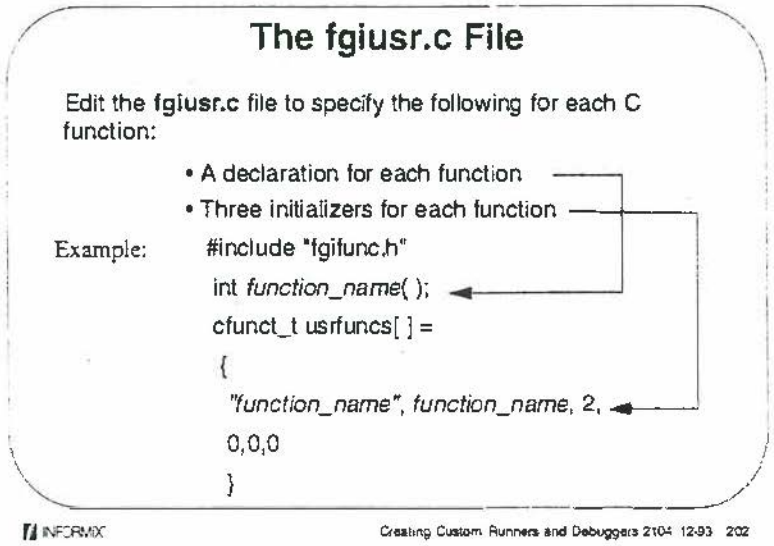

Creating a Custom Runner
There are two steps in creating a custom runner:
1. Edit a structure definition file called fgiusr.c to declare your C functions. This file is located in your SINFORMIXDIR/etc directory.
2. Compile and link the fgiusr.c file with the files that contain your C functions. This is done by using the cfglgo command.
INFORMIX
Creating Custom Runners and Debuggers 2104 12-93 201

Creating a Custom Runner

The fgiusr.c file is provided with the INFORMIX-4GL product and is located in your INFORMIXDIR/etc directory. The fgiusr.c file is a C language file that you can edit to declare programmer defined C functions.

Once you have declared your C functions in the fgiusr.c file, you can use the cfglgo command to create a custom runner. The cfglgo command compiles and links the edited fgiusr.c file with your C program files into an executable program.

Note: You must copy the fgiusr.c file to your current directory or specify the full path name.



The fgiusr.c File

For each C function in the **fgiusr.c** file, you must specify the following:

- A declaration for each function:
  int *function_name* ( );
- Three initializers for each function:
  "function_name", *function_name*, [-] integer,

The symbols ( ); must follow *function_name* in the declaration.

Initializers

The first initializer is a character pointer, the function name between quotation marks.

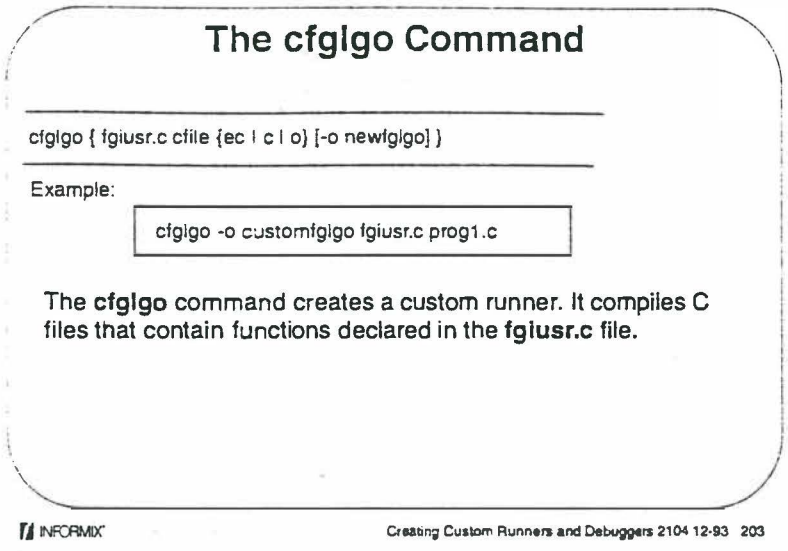
The second initializer is a function pointer, the function name without quotation marks.

The third initializer is an integer for the number of arguments expected by the function. If the number of arguments can vary, make the integer the maximum number of arguments, prefixed with a minus sign (-).

In the example above, the 4GL program will call a single C function called *function_name* with two arguments.

Note:

The file fgifunc.h and structure array usrfuncs are always included and already exist in the fgiusr.c file. The line of three zeros indicates that there are no more functions.


The cfglgo Command
cfglgo { fgiusr.c cfile {ec | c | o} [-o newfglgo] }
Example:
cfglgo -o customfglgo fgiusr.c prog1.c
The cfglgo command creates a custom runner. It compiles C files that contain functions declared in the fgiusr.c file.
INFORMIX
Creating Custom Runners and Debuggers 2104 12-93 203

The cfglgo Command

The **cfglgo** command compiles and links the edited **fgiusr.c** file with your C program files into an executable program that can be run with the custom runner.

You can rename the edited structure definition file. Specify the new file name in place of fgiusr.c.

If you do not specify the -o newfglgo option, the new custom runner is given the default name a.out. It is recommended to give the new runner a unique name.

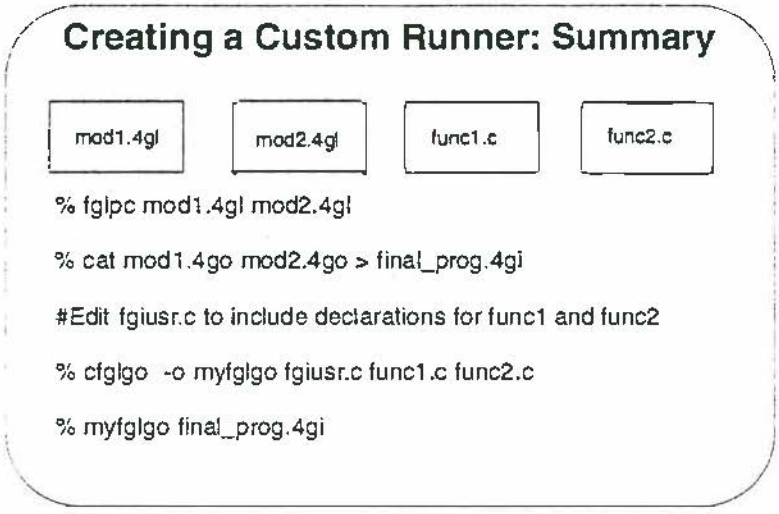
You can specify any number of compiled or uncompiled C files on the cfglgo command line.

If the fgiusr.c file is not in the current directory, you must specify the full path name.

After you create a customized runner, you can use it to execute any 4GL program whose C functions you correctly specified in the edited **fgiusr.c** file.

Makefiles

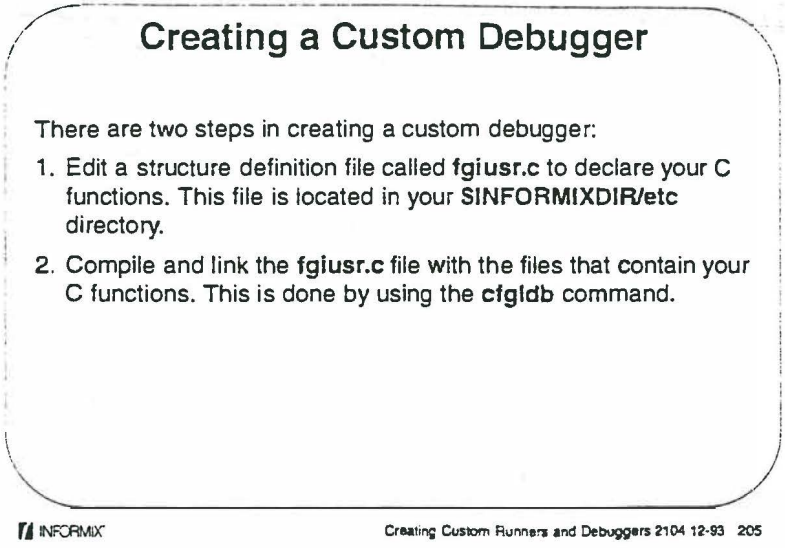
You can define a **makefile** to build your custom runners.



Creating a Custom Runner: Summary

The example above illustrates the following steps:

1. Compile **mod1.4gl** and **mod2.4gl** using the **fglpc** command. The **fglpc** command compiles each module into a **.4go** file that contains tables of information and blocks of p-code.
2. Concatenate the **.4go** files into a single file using the UNIX **cat** command.
3. Edit the **fgiusr.c** file to include C function declarations for **func1.c** and **func2.c**.
4. Create a custom runner using the **cfglgo** command. The **fglgo** command compiles your C files that contain functions declared in the **fgiusr.c** file.
5. Execute the program using your custom runner.


Creating a Custom Debugger
There are two steps in creating a custom debugger:
1. Edit a structure definition file called fgiusr.c to declare your C functions. This file is located in your SINFORMIXDIR/etc directory.
2. Compile and link the fgiusr.c file with the files that contain your C functions. This is done by using the cfgldb command.
INFORMIX
Creating Custom Runners and Debuggers 2104 12-93 205

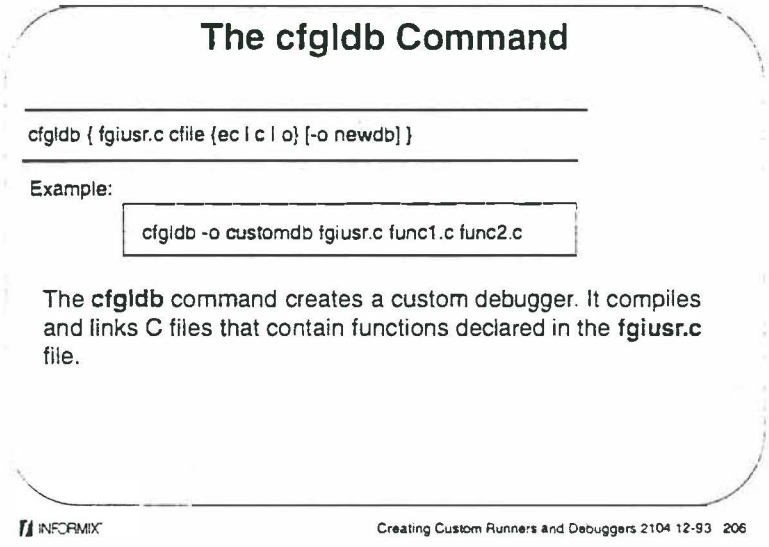
Creating a Custom Debugger

The fgiusr.c File

The **fgiusr.c** file is provided with the INFORMIX-4GL product and is located in your **SINFORMIXDIR/etc** directory. The **fgiusr.c** file is a C language file that you can edit to declare programmer defined C functions.

This is the same editing step required in creating a custom runner.

Once you have declared your C functions in the **fgiusr.c** file, you can use the **cfgldb** command to create a custom debugger for use with the applications that call your C functions. The **cfgldb** command compiles and links the edited **fgiusr.c** file with your C program files.


The cfgldb Command
cfgldb { fgiusr.c cfile {ec | c | o} [-o newdb] }
Example:
cfgldb -o customdb fgiusr.c func1.c func2.c
The cfgldb command creates a custom debugger. It compiles and links C files that contain functions declared in the fgiusr.c file.
INFORMIX
Creating Custom Runners and Debuggers 2104 12-93 206

The cfgldb Command

The cfgldb command compiles and links the edited fgiusr.c file with your C program files that can be run with a custom debugger.

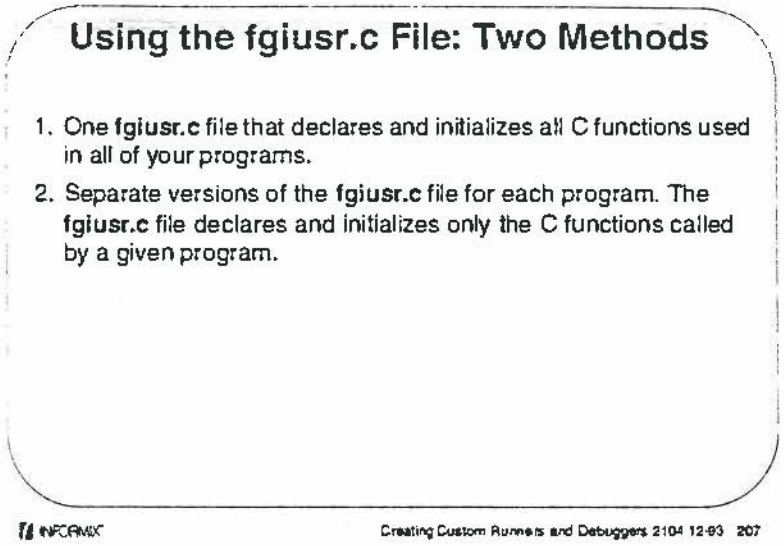
You can rename the edited structure definition file. Specify the new file name in place of fgiusr.c.

If you do not specify the **-o** newdb option, the new custom debugger is given the default name **a.out**. It is recommended to give the new debugger a unique name.

You can specify any number of compiled or un-compiled C files on the cfgldb command line.

If the **fgiusr.c** file is not in the current directory, you must specify the full path name.

After you create a customized debugger, you can use it to run the debugger with any 4GL program whose C functions you correctly specified in the edited fgiusr.c file.


Using the fgiusr.c File: Two Methods
1. One fgiusr.c file that declares and initializes all C functions used in all of your programs.
2. Separate versions of the fgiusr.c file for each program. The fgiusr.c file declares and initializes only the C functions called by a given program.
INFORMIX
Creating Custom Runners and Debuggers 2104 12-93 207

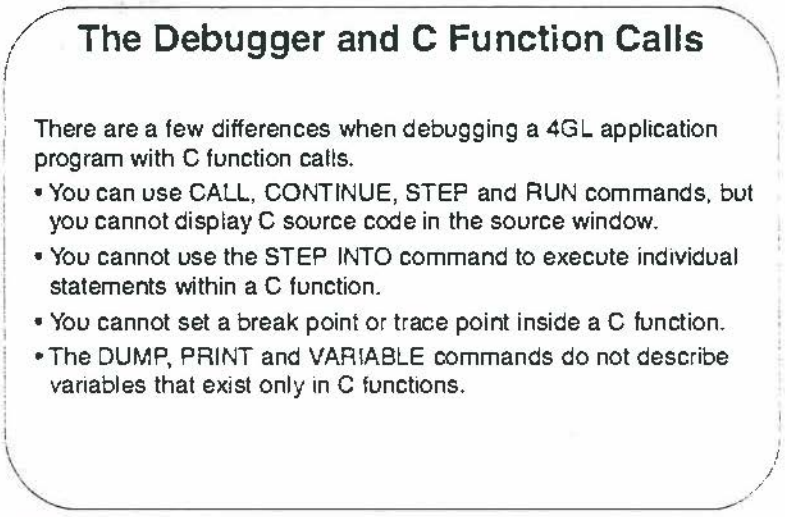
Using the fgiusr.c File: Two Methods

If you are developing several 4GL programs that call C functions, you can use the fgiusr.c file in one of two ways:

- Edit the fgiusr.c file and declare all of the C functions called in all of the programs. You can create a single custom runner and custom debugger to be used by all of the developers.
- Create different versions of the fgiusr.c file and declare only the C functions called in a single program. However, your developers must know which custom runner and debugger to use with each application. Each custom runner and debugger should have unique names.

The custom runner and debugger should be stored in a centralized location to provide access to all developers. In order to execute the custom runner or debugger, the location of both needs to be included in your path.

Note: A **makefile** can be defined to create your custom debuggers.


The Debugger and C Function Calls
There are a few differences when debugging a 4GL application program with C function calls.
• You can use CALL, CONTINUE, STEP and RUN commands, but you cannot display C source code in the source window.
• You cannot use the STEP INTO command to execute individual statements within a C function.
• You cannot set a break point or trace point inside a C function.
• The DUMP, PRINT and VARIABLE commands do not describe variables that exist only in C functions.
INFORMIX
Creating Custom Runners and Debuggers 2104 12-93 208

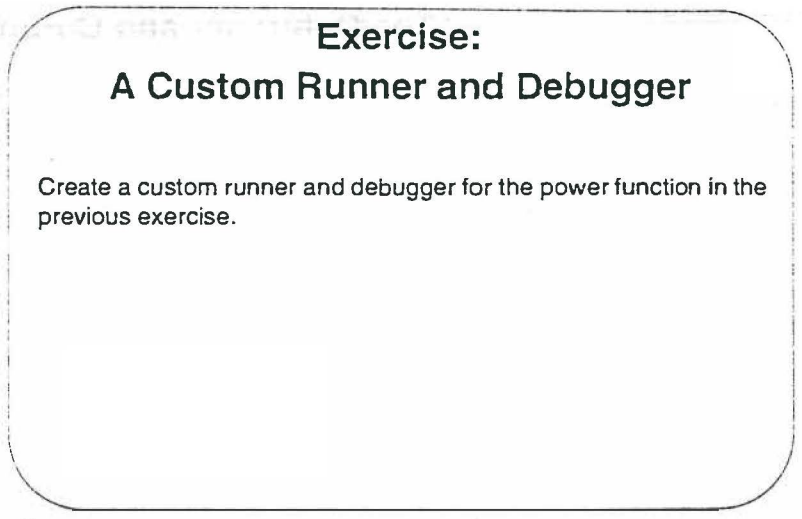
The Debugger and C Function Calls

There are a few differences when using the debugger with a 4GL program that has C function calls. They do not affect how the debugger interacts with 4GL statements. They do affect how the debugger interacts with the C programming statements.

The most important differences are:

- You can use CALL, CONTINUE, STEP and RUN commands, but you cannot display C source in the source window. To view the C source code, you must escape to the shell.
- You cannot use the STEP INTO command to execute individual statements within a C function. The 4GL statement that calls the C function and the C function are treated as a single statement by the STEP command.
- You cannot set a break point or trace point inside a C function.
- The DUMP, PRINT and VARIABLE commands do not describe variables that exists only in C functions.

The debugger allows you to trace source code up to the C function call and after the C function call, but not inside the C function.


Exercise:
A Custom Runner and Debugger
Create a custom runner and debugger for the power function in the previous exercise.
INFORMIX
Creating Custom Runners and Debuggers 2104 12-93 209

Exercise: A Custom Runner and Debugger

Create a custom runner and debugger for the **power** function in the previous exercise. You should be in your **$BASEDIR/src/exercise3** directory.

Step 1 — Compile a p-code version of your **stub.4gl** file using the **fglpc** command.

Step 2 — Copy the **fgiusr.c** file into the exercise3 directory. Edit the file to define your C function.

Step 3 — Create a custom runner using the **cfglgo** command. You need to include a **-lm** flag at the *end* of the **cfglgo** command line to include the mathematical library in C. Name the custom runner **customfglgo**.

Step 4 — Create a custom debugger using the **cfgldb** command. You need to include a **-lm** flag at the *end* of the **cfgldb** command line to include the mathematical library in C. Name the custom debugger **customfgldb**.

Step 5 — Run the program using the custom runner.

Step 6 — Invoke the ID using the custom debugger.

Solutions

Step 1

```
% fglpc stub.4gl
```

This creates an output file called **stub.4go**. If you had multiple modules in your application, you would concatenate each .4go into a single file with a **.4gi** extension.

```
% cat stub.4go module.4go > final_prog.4gi
```

Step 2

```
% cp $INFORMIXDIR/etc/fgiusr.c $BASEDIR/src/exercise3/fgiusr.c
#include "fgicfunc.h"

int pwr();

cfunc_t usrcfuncs[] =
 {
 "pwr", pwr, 2,
 0, 0, 0
 };
```

Step 3

```
% cfglgo  fgiusr.c  pwr.c  -o customfglgo  -lm
```

Step 4

```
% cfgldb  fgiusr.c  pwr.c  -o customdb  -lm
```

Step 5

```
% customfglgo  stub.4go 2 2
```

Step 6

```
% customdb  stub.4go
```

INFORMIX

Chapter

4GL Optimization

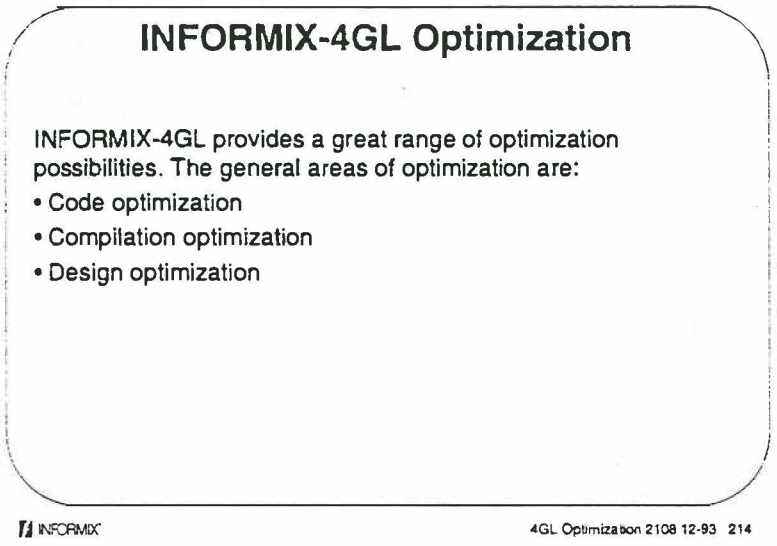
10

Objectives

At the end of this module, you will be able to:

- Explain the different memory requirements between INFORMIX-4GL C Compiler Version and INFORMIX-4GL Rapid Development System
- Use code optimization techniques and understand the way commands are used and implemented
- List compile optimization options which can contribute to more efficient executable code
- Design optimization strategies and consider the way you choose to structure your code

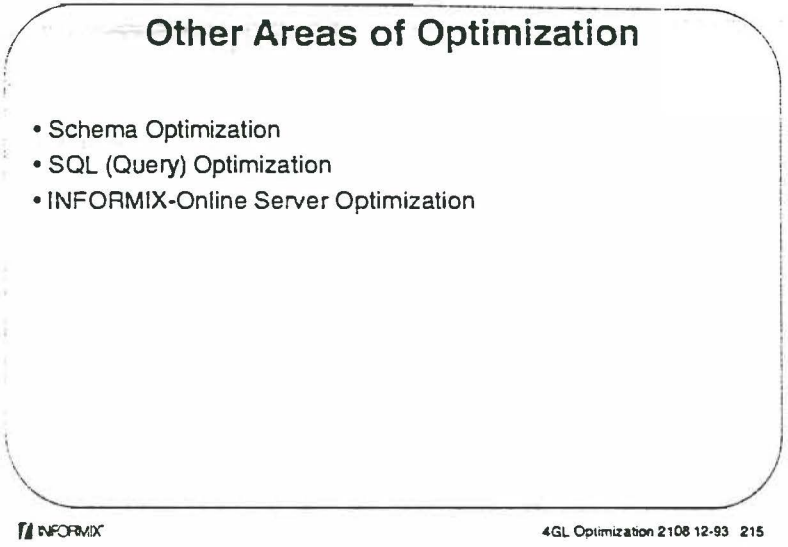
© 1993 Informix Software, Inc.


INFORMIX-4GL Optimization
INFORMIX-4GL provides a great range of optimization possibilities. The general areas of optimization are:
• Code optimization
• Compilation optimization
• Design optimization
INFORMIX
4GL Optimization 2108 12-93 214

INFORMIX-4GL Optimization

INFORMIX-4GL provides many areas of optimization possibilities. Some performance techniques will result in memory consumption. These optimization techniques are applicable to particular applications and environments. These general areas are broken into the following categories:

- Code optimization - considers the 4GL code, and understanding the way commands are used and implemented.
- Compilation optimization - covers command line arguments to the compilers which can contribute to more efficient executable code.
- Design optimization - considers the way you choose to structure your code. More importantly, it covers the way you choose to implement a particular feature, and when you choose not to use a feature to benefit performance.



Other Areas of Optimization

There are additional areas of optimization that are beyond the scope of this course. However, they are equally important in the success of your application and worth noting for your consideration:

- Schema Optimization
  - Recognizing and understanding the costs and benefits of indexing
  - Effectively indexing appropriate columns
  - Determining the level of data normalization in your schema
- SQL (Query) Optimization
  - Understanding the cost-based optimizer
  - Understanding ways to optimize a query
  - Using SET EXPLAIN
  - Using UPDATE STATISTICS
- INFORMIX-OnLine Server Optimization

Informix offers the following training courses that cover these issues, or consult your Educational Services Course Brochure:

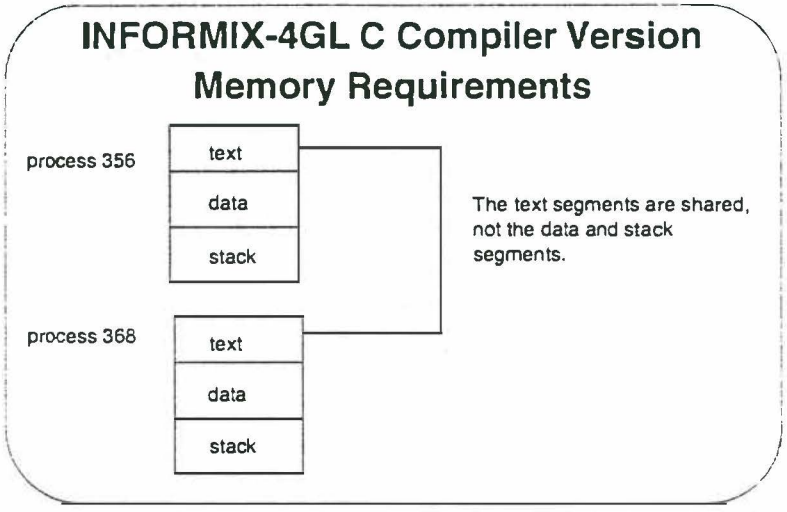
Relational Database Design

Structured Query Language

Online System Administration

Managing and Optimizing Informix Databases

Managing Large Databases



INFORMIX-4GL C Compiler Version Memory Requirements

UNIX Processes

It is important to understand the memory requirement differences between INFORMIX-4GL C Compiler Version and INFORMIX-4GL Rapid Development System. In general, compiled applications will use less memory than the same application executed using INFORMIX-4GL Rapid Development System.

Compiled applications, like all UNIX processes, consist of a text, data and stack segment. The text segments are shared, so a single copy exists in memory, regardless of the number of users executing the application. The extra memory per application is what is required for the data and stack space.

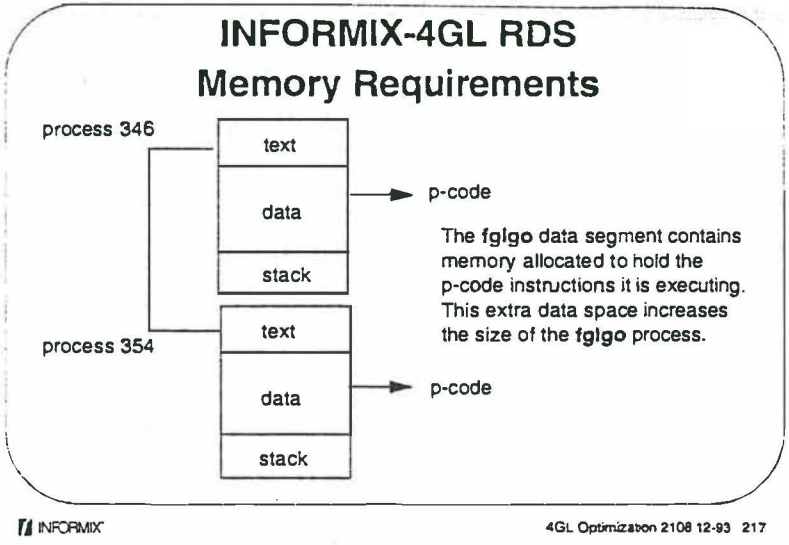
For any single application, the total memory requirements can be calculated by the formula:

*1 text segment + (users * (data + stack))*

The data and stack requirements for each individual process will differ slightly depending on what the process is asked to do. This difference is not enough to affect the calculation.

The UNIX Size Command

The size of an application's text segment and the initial size of its data segment can be determined by executing the UNIX **size** command on the application binary.



INFORMIX-4GL RDS Memory Requirements

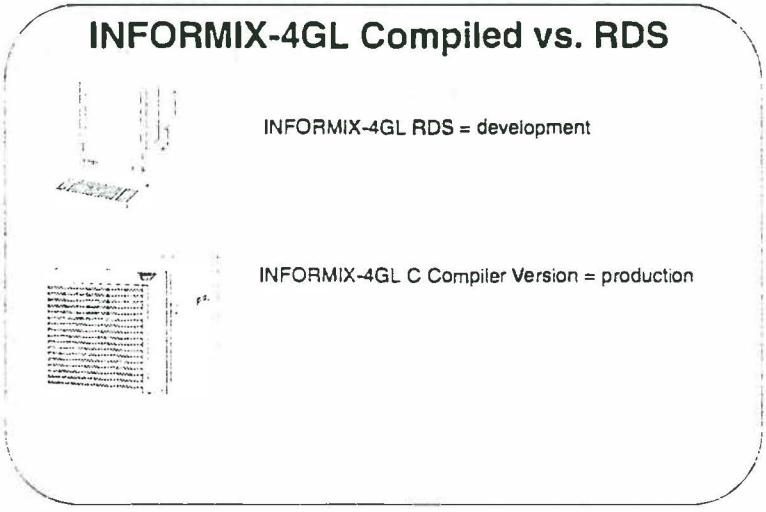
UNIX Processes

INFORMIX-4GL Rapid Development System (RDS) applications operate quite differently than INFORMIX-4GL C Compiler Version applications. The 4GL source code is compiled into a p-code file that is in turn interpreted by a p-code runner for execution.

The p-code runner is like a normal UNIX process and consists of a text, data and stack segment. The text segments are shared, so a single copy exists in memory, regardless of the number of users executing the application.

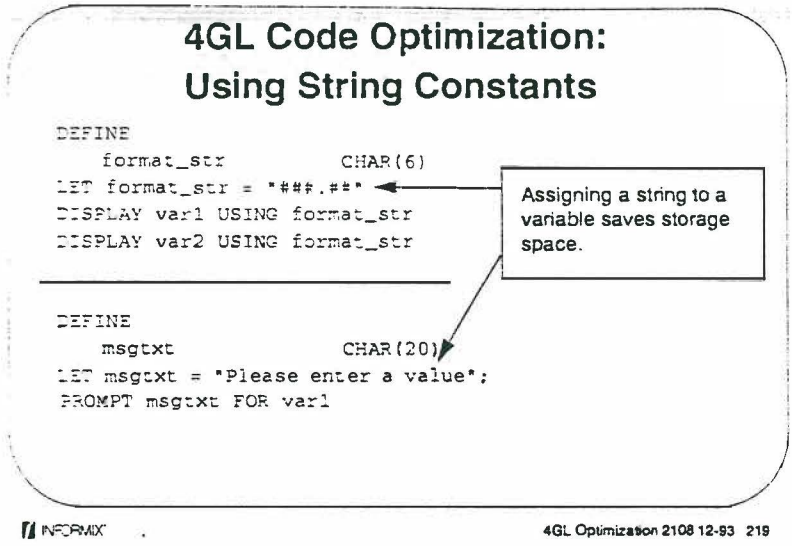
The memory requirement for each INFORMIX-4GL Rapid Development System application includes the data and stack segments of the p-code runner, as well as the memory allocated for the interpreted p-code itself. When the p-code runner begins execution, it allocates enough memory in the data segment to hold all of the interpreted p-code.

It is the extra memory required for the p-code itself that causes RDS to consume more memory than a compiled 4GL application.


INFORMIX-4GL Compiled vs. RDS
INFORMIX-4GL RDS = development
INFORMIX-4GL C Compiler Version = production
INFORMIX
4GL Optimization 2108 12-93 218

INFORMIX-4GL Compiled vs. RDS

INFORMIX-4GL C Compiler Version should be used in production systems to conserve memory and make efficient use of the CPU. The strength of INFORMIX-4GL Rapid Development System is in the development process, as it provides time savings and the ability to perform interactive debugging.



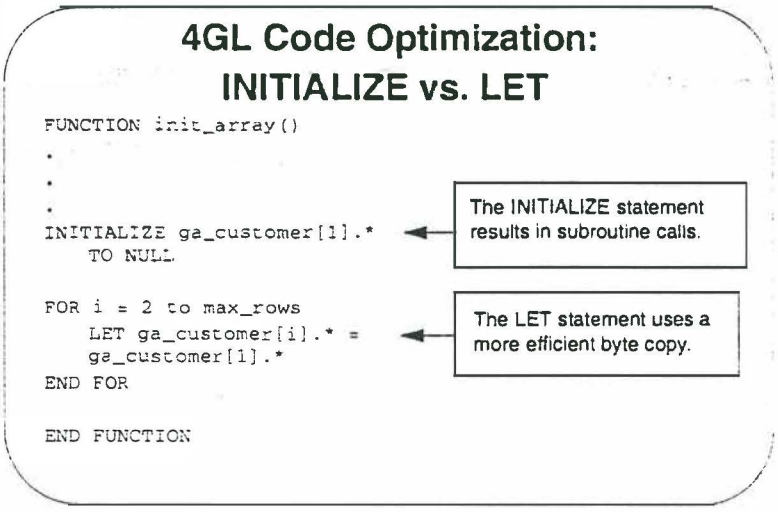
4GL Code Optimization: Using String Constants

There are static strings used in 4GL programs. These strings are enclosed by quotation marks. They are used for query strings, output messages and format strings. These strings are stored in a special area of memory that is part of the 4GL program. The area has a limited size, and if you use a large number of these strings, you can run out of space to store them. In addition, the program can become very large when a lot of space is required to store these strings. Reducing the amount of space needed for the strings can make your program more efficient.

To reduce string space, examine your program for common strings. Assign common strings to a variable and reference the variable. For example, you may have a commonly used format string like ###.##. Rather than repeat the quoted string in many USING clauses, assign the string to a variable and use the variable. The same method can be applied to messages. Keeping the size of the string variable to the exact size of the string will also cut down on wasted memory space.

Using a Database Table

Another method to reduce string space is to store the strings in a database table. The appropriate string can then be selected when needed. The disadvantage to this method is the additional database access required to SELECT the string.

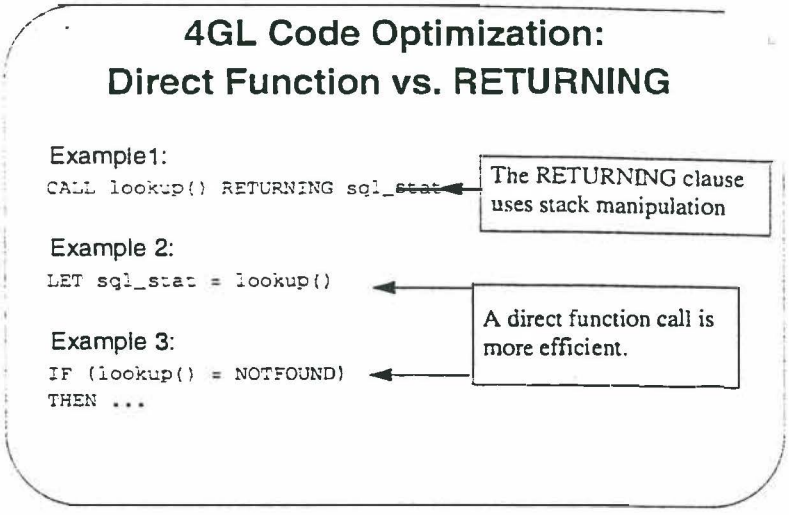


4GL Code Optimization: INITIALIZE vs. LET

The INITIALIZE and LET statements can be used to assign values to program variables. The INITIALIZE statement results in a series of subroutine calls to make the assignment. The LET statement uses a more efficient byte copy.

A single INITIALIZE of a program variable may not generate performance concerns. The performance problems arise when initializing large arrays. For efficiency, initialize one array record to NULL and use the LET statement to copy the NULL record.

This may improve your run-time performance and may decrease the size of your program.



4GL Code Optimization: Direct Function vs. RETURNING

You can call a function that returns a single value many different ways. The most common methods are:

- Use the RETURNING clause.
- Assign the returned value to a variable using the LET statement.
- Call the function directly within the expression when the returned value is needed only in the expression.

The RETURNING Clause

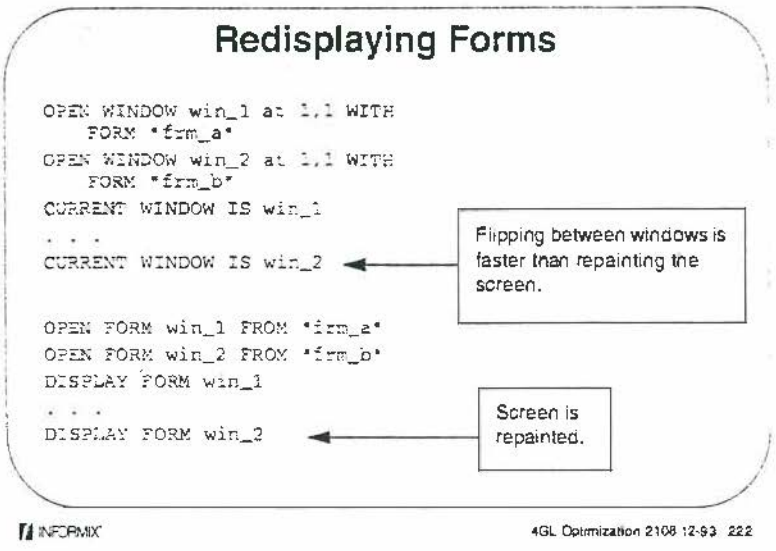
The RETURNING clause involves more stack manipulation than the other methods. The RETURNING clause is the most costly method and must be used in situations where multiple values must be returned.

The LET Statement

The LET statement is the preferred method when you need to use the returned value more than once. The LET statement reduces the amount of stack manipulation needed to move the returned value into the variable. Assigning the result to a variable is more efficient than repeated calls to the function.

Direct Function Call Within an Expression

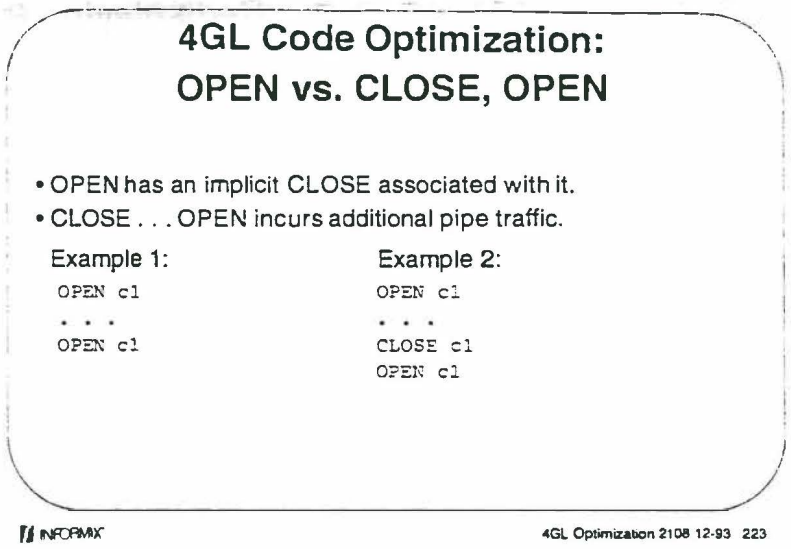
A direct function call within an expression is the best method, involving the least amount of stack manipulation. This method does assume that the resulting value is needed only once.



Redisplaying Forms

If you have multiple pages of input, use windows that are full screen and flip between them using the **CURRENT WINDOW IS** statement rather than repainting the current screen.

If you have a full page of input or output and need to move between pages, keep the pre-painted page in memory and display it by executing the **CURRENT WINDOW IS** statement. This method is faster than clearing the screen and repainting it with **DISPLAY** statements which are subject to the overhead of positioning and formatting.



4GL Code Optimization: OPEN vs. CLOSE, OPEN

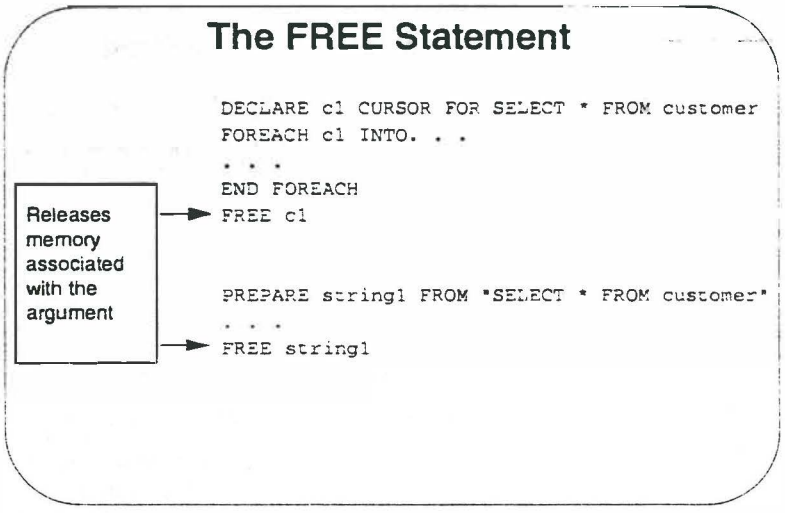
The OPEN statement is used to establish the search criteria (via the USING clause) for a SELECT statement and to initialize the system for subsequent FETCH operations. Or it is used to establish an insert buffer for an INSERT cursor that references program variables.

The OPEN statement allocates the data buffer to the application process and the cursor block to the database server process. A CLOSE statement releases resources allocated by the OPEN statement.

If a SELECT cursor is already open, an OPEN statement closes the cursor and reopens it, creating a new active set based upon the current value of the program variables in the USING clause.

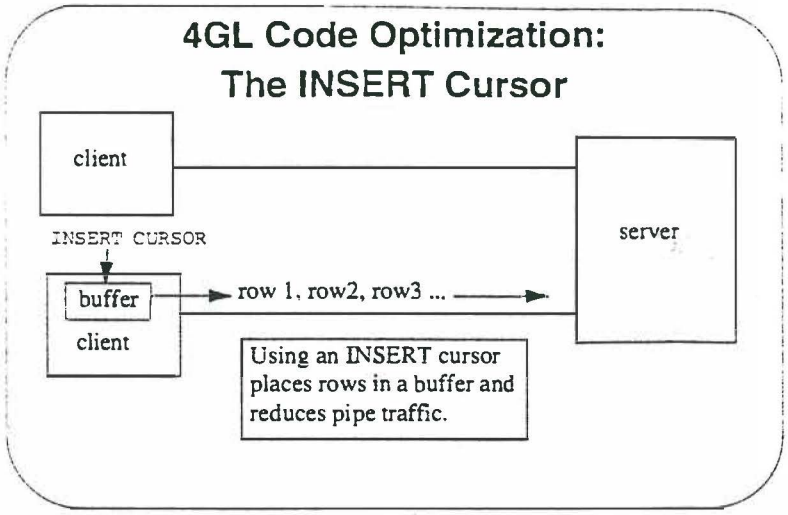
Therefore, if you need to reopen a currently opened cursor, you can omit the CLOSE statement. It provides no additional functionality and will cause additional pipe traffic between the client and server processes. However, you may need to explicitly close your cursors for memory optimization.

Note: This technique is not recommended with INSERT cursors.



The FREE Statement

The FREE statement can take two types of arguments: a *statement ID* or *cursor name*. The FREE statement releases database server resources allocated by the **PREPARE** (statement ID) or **DECLARE** (cursor) statements. These resources include memory. Freeing memory allows the server to reuse the memory space for other statements requiring memory, which helps keep the size of the server process as small as possible.



4GL Code Optimization: The INSERT Cursor

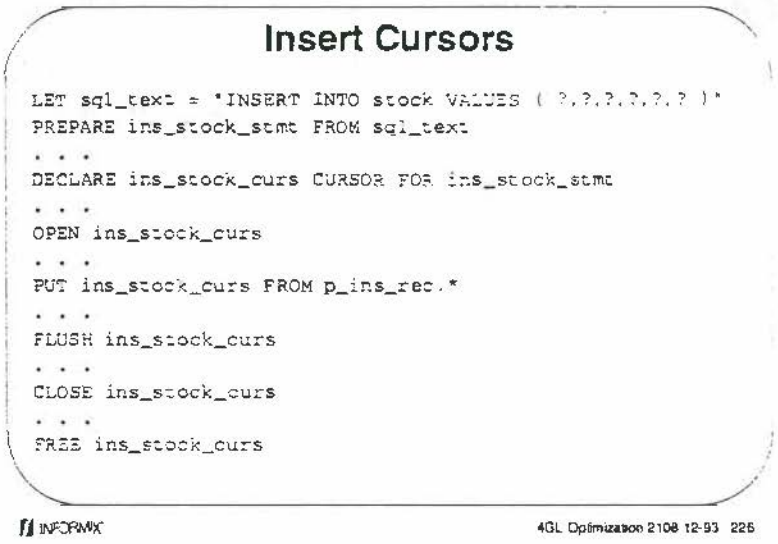
To INSERT a single row into a table, the client and server must exchange information over pipes. If the application is distributed over multiple machines using INFORMIX-NET or INFORMIX-STAR, the exchange must take place over a network. This exchange could be especially slow over a network. In either case, using INSERT cursors to reduce communications between the client and server can help your application run faster.

When you declare an INSERT cursor, a buffer is created in memory of the client process to hold the rows you want to insert. You can add rows to the buffer using the PUT command. When the buffer is full it will automatically be flushed to disk. You can manually flush the buffer with the FLUSH command or by closing the cursor.

Disadvantages

If an error occurs and the row fails to be inserted into the database, all rows after the unsuccessful row will also fail. Status information for the PUT, FLUSH or CLOSE statements can be obtained from the **SQLCA.SQLERRD[3]** variable.

The user can also lose data in the buffer if the cursor is not closed properly (following a user interrupt), or a system failure occurs. INSERT cursors are commonly used in batch processes. They are recommended only if the data can be validated to maximize the success of the inserts.

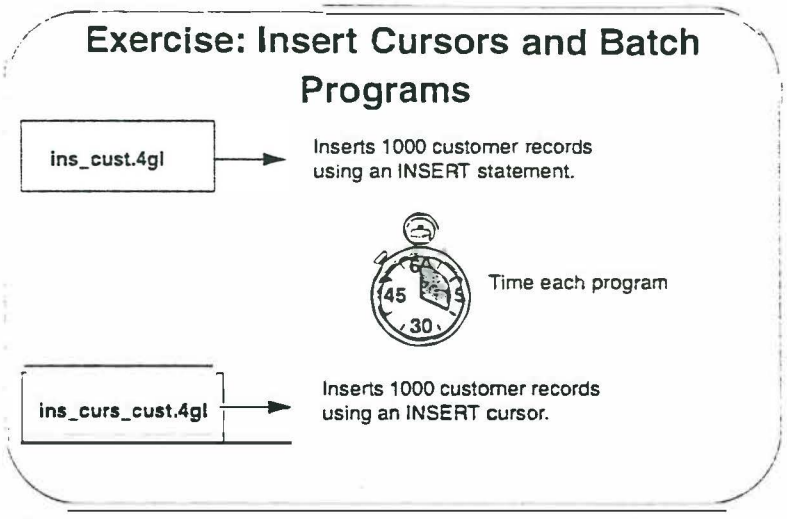


Insert Cursors

The following statements allow you to declare and manipulate an INSERT cursor:

DECLARE	associates a cursor with an INSERT statement.
OPEN	creates an insert buffer for the INSERT cursor.
PUT	places a row in the insert buffer. When the buffer is full, INFORMIX-4GL automatically sends the rows to the server for insertion into the appropriate table.
FLUSH	inserts the rows into the database tables without closing the cursor.
CLOSE	flushes the insert buffer and closes the INSERT cursor.

If you are using a database with transaction logging, the OPEN, PUT, FLUSH and CLOSE statements must be inside a transaction.


Exercise: Insert Cursors and Batch Programs
ins_cust.4gl
Inserts 1000 customer records using an INSERT statement.
Time each program
ins_curs_cust.4gl
Inserts 1000 customer records using an INSERT cursor.
INFORMIX
4GL Optimization 2108 12-93 227

Exercise: Insert Cursors and Batch Programs

Create two single module batch programs to insert 1000 customer records into the **customer** table. One program will insert the customer records using an INSERT statement; the other will use an INSERT cursor. You will time each program using the UNIX **time** command. You should be in your $BASEDIR/src/exercise4 directory.

Step 1 Create a module named ins_cust.4gl that uses a FOR loop to insert 1000 rows (of the same record) into the customer table using an INSERT statement.

Tip: The customer number is a SERIAL data type.

Step 2 Create a second module named ins_curs_cust.4gl that uses a non-prepared INSERT cursor and a FOR loop to insert 1000 rows (of the same record) into the **customer** table using the INSERT cursor.

Step 3 Compile, run and time each program using the UNIX **time** command:

```
% time program_name.4ge
```

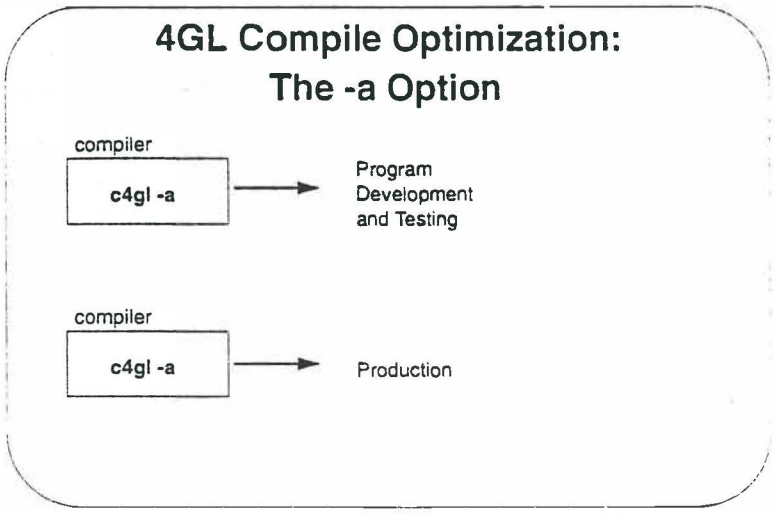
Solutions

ins_cust.4gl

```
DATABASE stores

MAIN

    DEFINE

        i SMALLINT

    FOR i = 1 TO 1000

        INSERT INTO customer VALUES ( 0, 'Dowling', 'Kalen',
            'KC Cycles', '123 El Camino', '', 'Palo Alto',
            'CA', 94303, '(415)555-1234' );

    END FOR

END MAIN
```

ins_curs_cust.4gl

```
DATABASE stores

MAIN

    DEFINE

        i SMALLINT

    DECLARE ins_cust_curs CURSOR FOR
    INSERT INTO customer VALUES ( 0, "Dowling", "Kalen",
        "KC Cycles", "123 El Camino", "", "Palo Alto",
        "CA", 94303, "(415)555-1234" );

    OPEN ins_cust_curs

    FOR i = 1 TO 1000

        PUT ins_cust_curs

    END FOR

    CLOSE ins_cust_curs

END MAIN
```

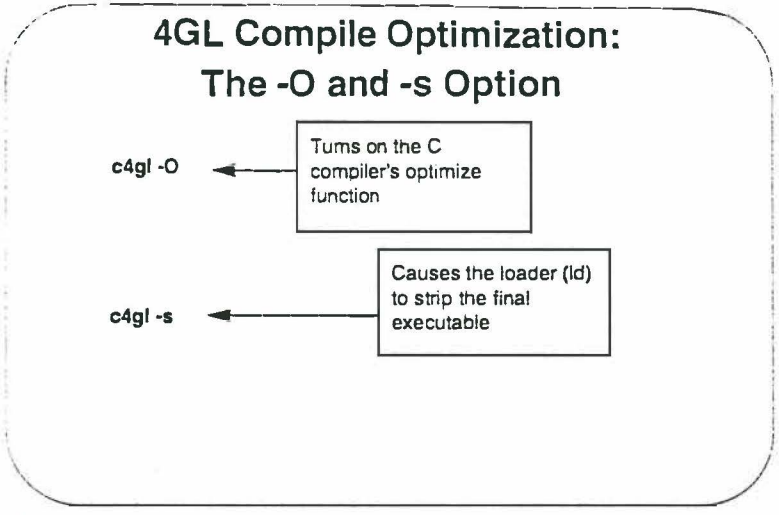


4GL Compile Optimization: The -a Option

The -a compile option specified at the **c4gl** command line causes the compiled program to check array bounds at run-time. This prevents you from exceeding the array bounds in your program. This checking is accomplished by adding additional code to your program.

Not For Production

This extra code is useful during the program development and debugging, but should not be used for production purposes. The extra code can slow down the performance of the program. Once your code is tested, recompile all modules without the **-a** compile option.


4GL Compile Optimization:
The -O and -s Option
c4gl -O
Turns on the C compiler's optimize function
c4gl -s
Causes the loader (ld) to strip the final executable
INFORMIX
4GL Optimization 2108 12-93 233

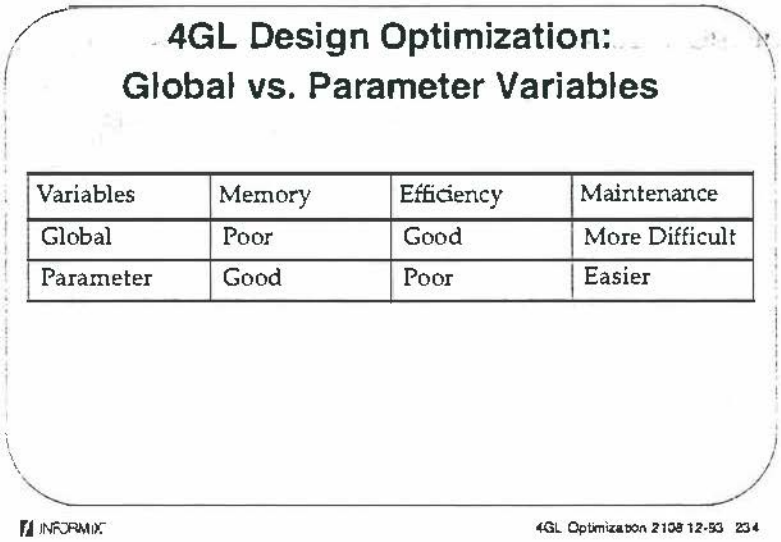
4GL Compile Optimization: The -O and -s Option

The -O compile flag is an option available to many C compilers. You can specify it at the **c4gl** command line to tell the C compiler to attempt to optimize the object code it produces. This will help produce the most optimal executable possible.

The -s compile flag is an option available to many C compilers. You specify it at the **c4gl** command line to tell the loader (ld) to strip the output it produces to save space.

The advantages with these two options are:

- to produce the most optimal executable code possible.
- to keep the executable size as small as possible. A smaller executable size means the program is less likely to be swapped out of memory during execution.


4GL Design Optimization:
Global vs. Parameter Variables
Variables | Memory | Efficiency | Maintenance
Global | Poor | Good | More Difficult
Parameter | Good | Poor | Easier
INFORMIX
4GL Optimization 2108 12-93 234

4GL Design Optimization: Global vs.

You have two methods of making data available to different functions performing operations:

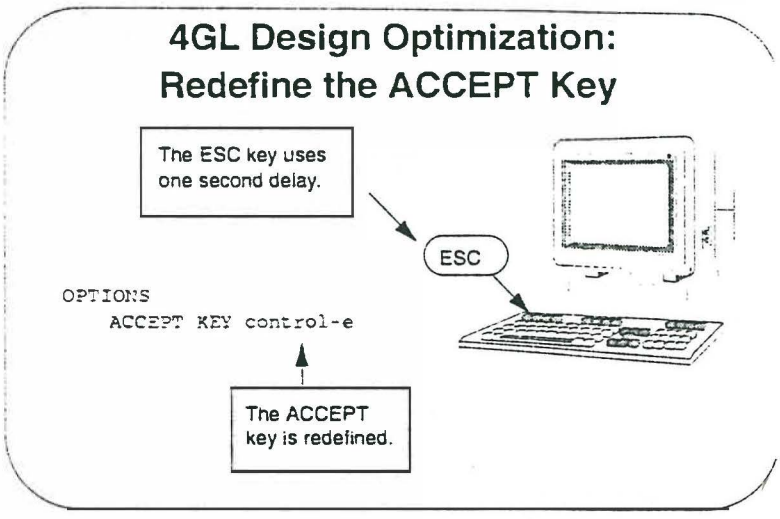
- Pass the data to the functions using parameter variables (local variables).
- Make the same copy of the data available to all functions by storing it in global variables.

Using global variables is the more efficient of the two methods. 4GL passes parameters via a stack. The calling function pushes the parameters onto the stack, and the called function pops the parameters off of the stack into local variables. The stack manipulation takes time that is saved when using global variables.

Disadvantages

Using global variables extensively will impact the memory required to run your program. The memory requirement for the global variables is repeated for each user running the program. It is also important to consider that global variables can make maintaining the code more difficult. If the global variable changes, you will have to propagate the change everywhere the variable is used. Also, global variables, if not handled correctly, are often the cause of logic errors and are difficult to debug.

Note: When considering optimization techniques, remember that module variables have the same internal memory requirements as global variables.



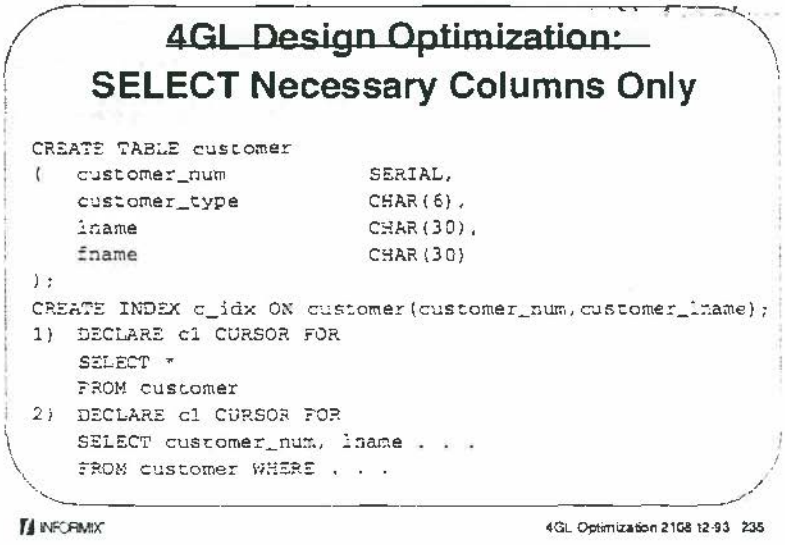
4GL Design Optimization: Redefine the ACCEPT Key

An INPUT statement can be terminated by an INTERRUPT or ACCEPT key. The default ACCEPT key is the ESC key.

Many keys, such as function keys, use escape sequences. In escape sequences, an escape is followed by one or more other values. 4GL I/O functions must be sure that when an escape is input, it is not part of an escape sequence. This check is performed by waiting about one second to see if any values follow the escape before processing the input as an escape. This delay can appear as slow performance to end users.

Control Keys

To avoid the delay, you can define your ACCEPT key to be something other than ESC. The preferred choice for an ACCEPT key is a CONTROL key. This can be redefined for all input statements using the OPTIONS command.



4GL Design Optimization: SELECT Necessary Columns Only

A feature of the Cost Based Optimizer Version 4.*x* is the ability to SELECT only the data columns from the data pages, if necessary. If all columns in the projection list are a usable index, the server will read only from the index pages; the data pages will not be read. A KEY ONLY read is the fastest and most efficient SELECT statement.

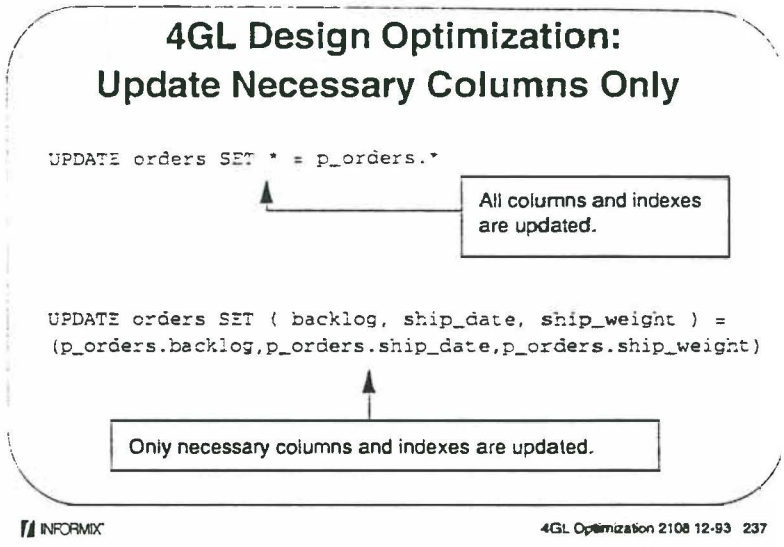
In the example above, if the customer number and last name columns are the only required columns, then select only those columns.

Two Process Architecture

The other benefit in selecting only necessary columns is realized with an understanding of the INFORMIX two-process architecture. The two-process architecture consists of the *client* (a 4GL program) and *server* (INFORMIX-OnLine).

The internal buffer used by the client is a relatively fixed size. The more data that is selected, the more context switches may be required to fetch the data from the server to the client. Filling the buffer involves pipe traffic between the client and server processes, and this can have a direct impact on the performance of the programs.

One way to reduce the amount of data is to select fewer rows. You should always strive to select only the rows needed. You can also reduce the amount of data by selecting fewer columns.


4GL Design Optimization:
Update Necessary Columns Only
UPDATE orders SET * = p_orders.*
All columns and indexes are updated.
UPDATE orders SET (backlog, ship_date, ship_weight) =
(p_orders.backlog,p_orders.ship_date,p_orders.ship_weight)
Only necessary columns and indexes are updated.
INFORMIX
4GL Optimization 2108 12-93 237

4GL Design Optimization: Update Necessary Columns Only

When you update a row in a table that is indexed, you will not only have to update the data in the data pages, you also have to update every index page containing an updated value. If you use the UPDATE command but do not change the value of the column, all indexes on that column will be altered.

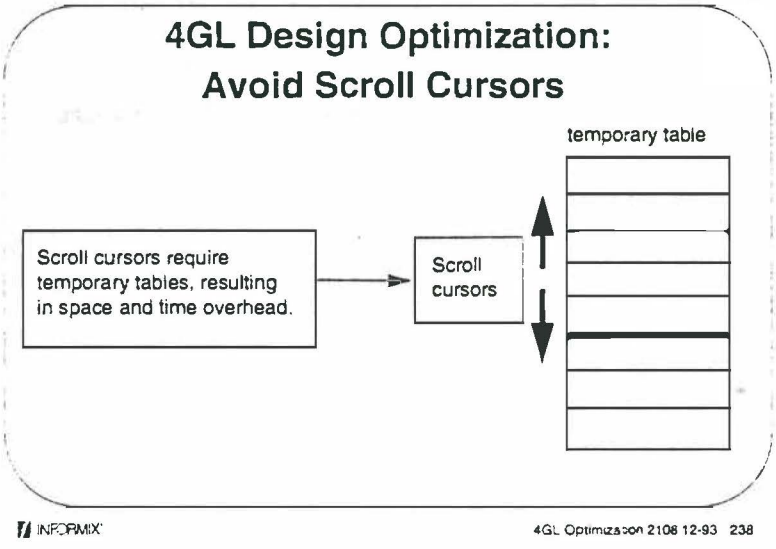
Specify only those columns that will be changed. It is not recommended to use the all inclusive asterisk syntax in an UPDATE statement unless all columns are affected.

Avoid Updating Key Values

Whenever possible, avoid updating columns corresponding to index keys. Updating a key value involves:

1. searching the index B+tree for the current location of the key.
2. deleting that entry from the index.
3. searching for the new location in the index.
4. inserting the new key value.

This process is repeated for every index that is part of the column. The updating of indexes can take time; the more indexes affected, the more time required for the update.



4GL Design Optimization: Avoid Scroll Cursors

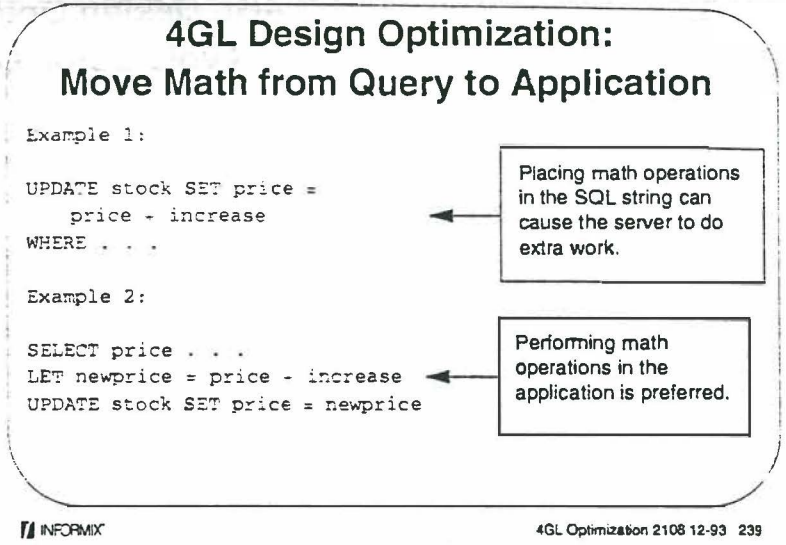
Scroll cursors are implemented by creating a temporary table containing the selected rows, through which the cursor can move forward or backwards.

Creating temporary tables incurs overhead in space and time: the time to process and write the selected rows to the temporary table, and space to hold the temporary table. This overhead can slow down your process. This is especially true when a large number of rows is selected.

When a large number of rows is selected, it can take a significant amount of time to select all the rows and write them to the temporary table. In the worst case, in a FETCH LAST or a SELECT with an ORDER BY, the entire temporary table must be created before the first FETCH will return.

Careful Consideration

You should carefully consider if a scroll cursor is really needed to implement the functionality required in your program. Never use a scroll cursor if it is not needed.


4GL Design Optimization:
Move Math from Query to Application
Example 1:
UPDATE stock SET price =
price + increase
WHERE . . .
Placing math operations in the SQL string can cause the server to do extra work.
Example 2:
SELECT price . . .
LET newprice = price + increase
UPDATE stock SET price = newprice
Performing math operations in the application is preferred.
INFORMIX
4GL Optimization 2108 12-93 239

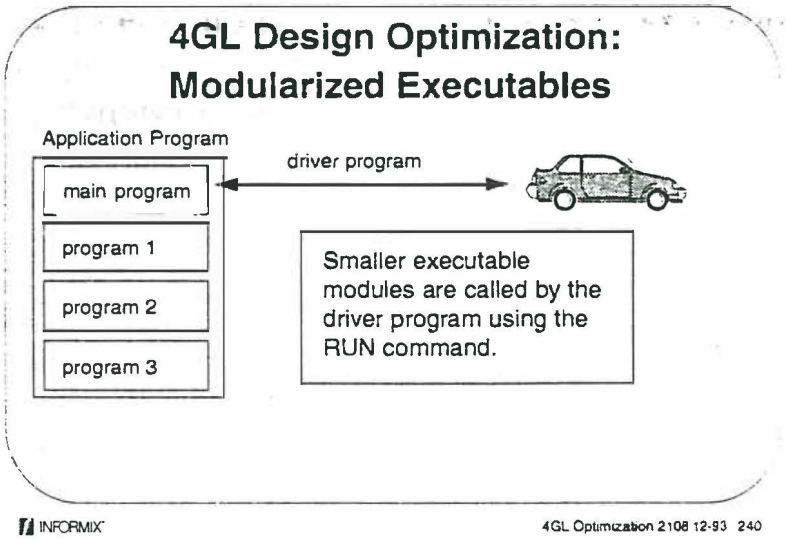
4GL Design Optimization: Move Math from Query to Application

Math calculations can be performed by either the client or server processes. It is common to place mathematical operations in the SQL queries. The argument against this approach is that the database server is not considered a calculation engine.

It is generally more efficient to keep math calculations that need to be performed on the data in the application. However, without careful evaluation, this method can result in increased pipe traffic and overly complicated 4GL code to perform the processing.

There are advantages to performing the math calculations in the server that you must consider:

- Letting the server perform the processing and returning only the results, viewed in context of the two-process architecture, the number of rows passed to the client may be greatly reduced.
- The 4GL code can be simplified.



4GL Design Optimization: Modularized Executables

Very large executable files can cause a program to run slower due to the process being swapped out of memory frequently. It is desirable to keep the executable programs as small as possible.

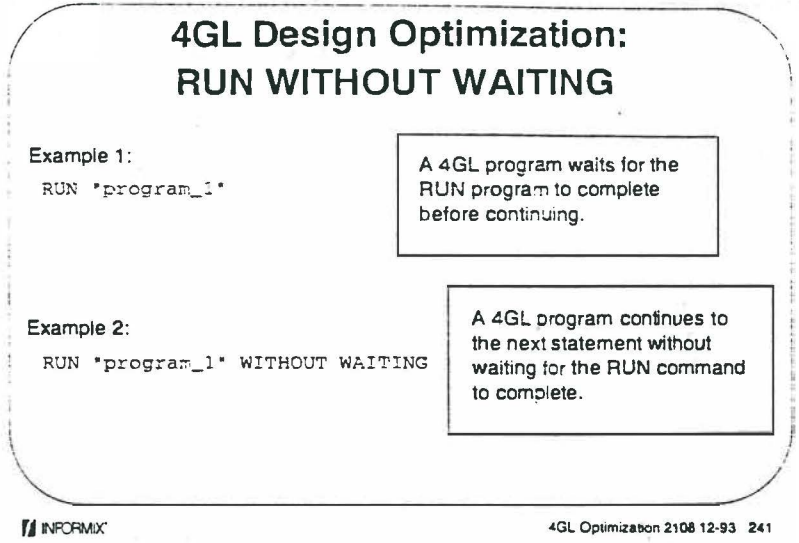
One method to reduce the size of the application program is to break it down into several smaller programs. The smaller programs are called from a *main* program, also considered a *driver* program. It is called a driver because it calls or *drives* the other programs.

A driver program can be written in 4GL, and can use the **RUN** command to execute the smaller programs. It can also be written using the UNIX shell language.

The RUN Command

The RUN command spawns a child process from your 4GL program. There are disadvantages to this approach:

- Overhead in managing the communication between programs.
- Multiple server processes. A new **sqlturbo** is spawned for each RUN command when execution reaches the DATABASE statement.
- Each executable must have the minimum requirements for a 4GL program (for example, the INFORMIX libraries).

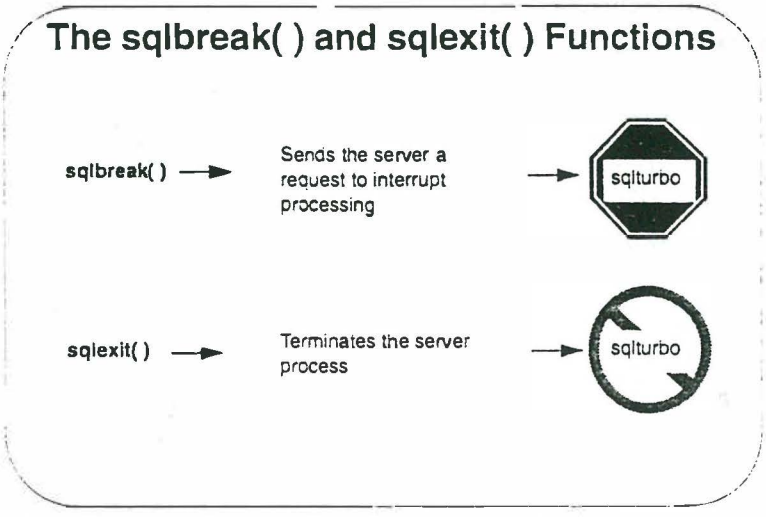

4GL Design Optimization:
RUN WITHOUT WAITING
Example 1:
RUN "program_1"
A 4GL program waits for the RUN program to complete before continuing.
Example 2:
RUN "program_1" WITHOUT WAITING
A 4GL program continues to the next statement without waiting for the RUN command to complete.
INFORMIX
4GL Optimization 2108 12-93 241

4GL Design Optimization: RUN WITHOUT WAITING

If you use the RUN command, the 4GL program waits for the RUN program to complete before continuing. If there is no need to wait for the RUN program to complete, then you should use the **WITHOUT WAITING** option. The 4GL program continues immediately to the next 4GL statement without waiting for the specified command to complete. By not waiting, your application runs faster.

RETURNING

If your program returns a value that you must save, use the RUN RETURNING *variable* option. The 4GL program waits until the run program is complete and puts the return value into the variable specified by the RETURNING keyword.



The sqlbreak( ) and sqlexit( ) Functions

The RUN command spawns your 4GL child process and an **sqlturbo** database server process. Each database server process can consume a significant amount of memory. There are various run-time SQL library functions that can help you manage your database server processes.

The **sqlbreak( )** library function sends the database server process a request to interrupt processing.

The **sqlexit( )** library function terminates the database server process, thereby freeing resources. It should be used to reduce database process overhead in programs that use the RUN command.

Both functions are available with the INFORMIX-4GL C Compiler Version. If you want to use the functions with the INFORMIX-4GL Rapid Development System, you must define the functions in the **fgiusr.c** file and create a custom runner.

Tip: The **sqlexit** function should only be called after issuing a CLOSE DATABASE statement. If **sqlexit** is called with an open database that uses transactions, it rolls back any current transactions and then closes the database.

Warning! sqlexit closes all open cursors and releases resources associated with prepared statements.

```
##############################################################
# Function : run_with_sql_exit
# Description : Prior to executing the 'RUN' statement on the
#               specified command, the current executables
#               database engine is terminated. Following
#               execution of the command, the
#               database engine for the once-again active
#               executable is opened and the appropriate database
#               environment is restored.
#
# Incoming Parameters : cmd - run command string to execute.
#                       db_name - current database name
#
# Return Parameters : <none>
# Impact Globals : <none>
# Perform Screens : <none>
#
# Author          : <name>
# Date            : <10/10/93>
# Modification Description: Original code.
##############################################################

FUNCTION run_with_sql_exit ( cmd, db_name )

    DEFINE
        cmd     CHAR(80),
        db_name CHAR(12)

    WHENEVER ERROR CALL err_func

 -- Release system resources by terminating the database engine.

    CALL sqlbreak()
    CALL sqlexit()

 -- Run the command specified.


    RUN cmd

 -- Reopen the current database and place the database in the
proper state.

    CALL sql_start_database ( db_nam )

END FUNCTION
```

INFORMIX®

Chapter

11

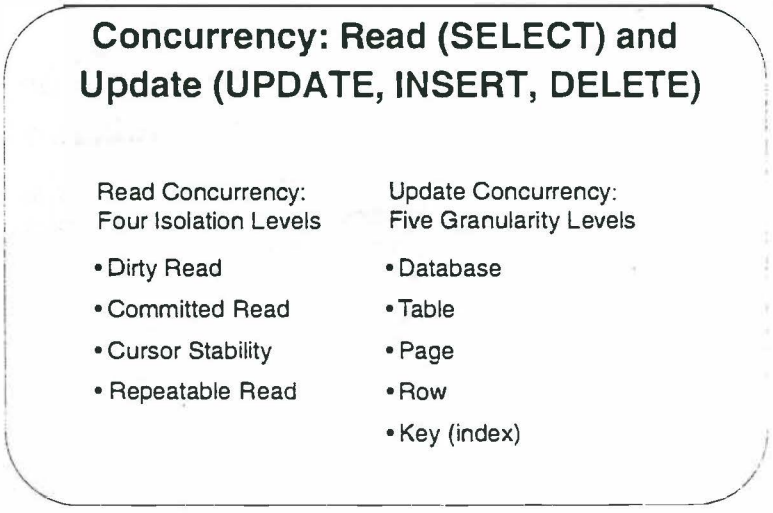
Locking, Transactions and Performance

Objectives

At the end of this module, you will be able to:

- Understand the two classes of concurrency
- List the types of locks the server uses to enforce concurrency
- Explain the four isolation levels
- Understand the different locking granularities
- Understand deadlocks
- Evaluate transaction designs and alternate locking strategies

© 1993 Informix Software, Inc.



Concurrency: Read (SELECT) and Update (UPDATE, INSERT, DELETE)

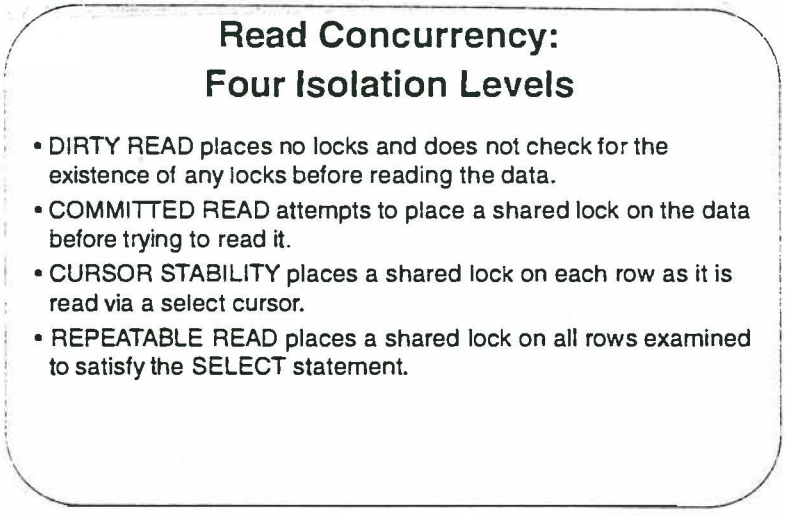
Concurrency refers to the cooperation between two or more processes in reading and updating data. There are two classes of concurrency control:

- Read concurrency applies to read-only access. Read concurrency is implemented by the use of isolation levels. INFORMIX-OnLine offers four levels of isolation. INFORMIX-SE only offers one level, dirty read.
- Update concurrency applies to changing database data. Update concurrency is implemented by acquiring locks at a particular granularity level. Granularity level refers to the size of the object being locked. There are five levels of granularity. INFORMIX-SE does not support page level granularity.

Types of Locks

Informix enforces concurrency control by using locks. There are three types of locks that can be used:

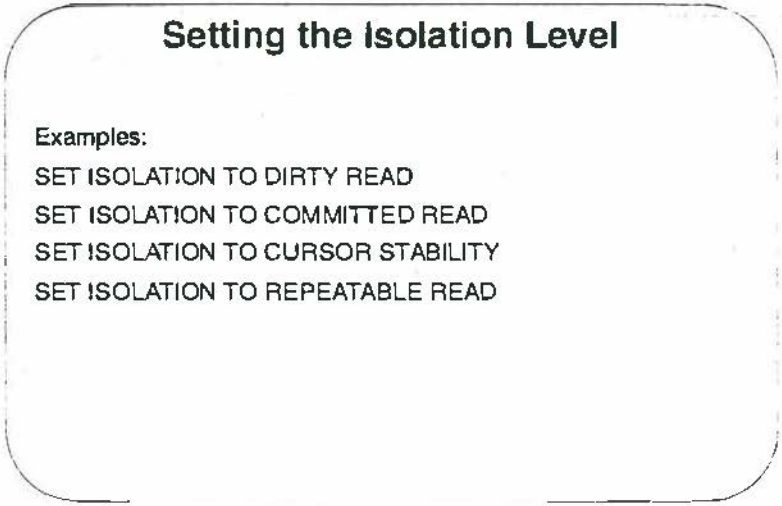
- *Exclusive locks* are used to manage *update* concurrency. No other locks can be placed on the data that holds an exclusive lock.
- *Shared locks* are used to manage *read* concurrency. They are placed by processes reading data. Multiple shared locks can be placed on a piece of data. Shared locks are only used by INFORMIX-OnLine.
- *Update (promotable) locks* are used to manage update concurrency when using update cursors. It is similar to a shared locked that can be promoted to an exclusive lock.


Read Concurrency:
Four Isolation Levels
• DIRTY READ places no locks and does not check for the existence of any locks before reading the data.
• COMMITTED READ attempts to place a shared lock on the data before trying to read it.
• CURSOR STABILITY places a shared lock on each row as it is read via a select cursor.
• REPEATABLE READ places a shared lock on all rows examined to satisfy the SELECT statement.
INFORMIX
Locking, Transactions and Performance 2107 12-93 247

Read Concurrency: Four Isolation Levels

Informix manages read concurrency through the use of isolation levels. These are implemented by the use of shared locks as follows:

- Dirty read does not provide any isolation for your process. During retrieval, you can look at any data, even rows containing uncommitted changes. Such rows are referred to as *dirty* data or *phantom* data.
- Committed read attempts to place a shared lock on the data before reading it. It does not place the lock, it only checks to see if it can acquire a lock. This assures you that the data exists at the time you read it and is not in the process of being updated by another process.
- Cursor stability places a shared lock on each row as it is read using a SELECT cursor. The shared lock is held until the next FETCH is executed. This assures you that the row exists and will continue to exist while you are looking at it. If a cursor is not used, cursor stability behaves in the same manner as committed read.
- Repeatable read places a shared lock on all rows examined to satisfy the query. This includes rows that do not meet the SELECT criteria, but had to be examined to determine their ineligibility. Additionally, the corresponding index keys are also locked. The locks are held until the transaction is committed.


Setting the Isolation Level
Examples:
SET ISOLATION TO DIRTY READ
SET ISOLATION TO COMMITTED READ
SET ISOLATION TO CURSOR STABILITY
SET ISOLATION TO REPEATABLE READ
INFORMIX
Locking, Transactions and Performance 2107 12-93 248

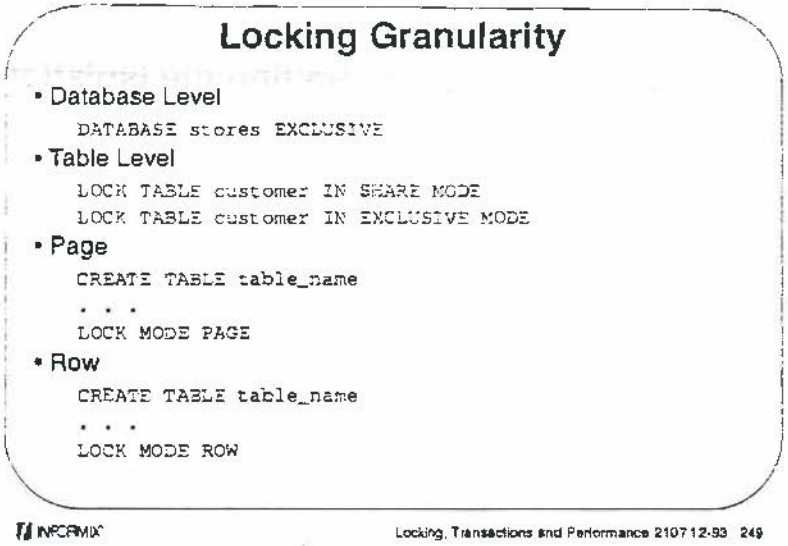
Setting the Isolation Level

To use process isolation, your database must use logging. You can use the SET ISOLATION statement to pick an isolation level. The default isolation levels are as follows:

- Database without logging - DIRTY READ
- Non ANSI-compliant database - COMMITTED READ
- ANSI-compliant database - REPEATABLE READ
- INFORMIX-SE database - DIRTY READ (only level)

Each client process can set its own isolation level. This can be implemented by setting an environment variable. The environment variable can be obtained in the 4GL program by the FGL_GETENV library function.

For maintainability, it is recommended to set the default isolation level for your program in your MAIN (or *driver*) function. If a different isolation level is required, it should be set and then reset to the default level within the function where it is used.



Locking Granularity

Locking granularity refers to the size of the data object being locked. The higher the granularity level, the lower the concurrency available to processes.

Database Level

Opening a database places a shared lock on the database. The entire database can be locked exclusively using the DATABASE statement with the EXCLUSIVE option. This opens and places an exclusive lock on the database, allowing access to the current user process only. You must execute the CLOSEDATABASE statement to allow other users access.

Table Level

You can lock an entire table in shared or exclusive mode. If your database uses transactions, tables must be locked within a transaction. A shared lock on the table allows other users read (SELECT) access only. An exclusive lock prevents other users from any access to the table. Only one lock is used to lock the table, regardless of the number of rows accessed within a transaction.

Unlocking a Table

The UNLOCK TABLE statement restores access to a previously locked table.

Row and Page Level

Page and row level locking is determined at table creation time using the LOCK MODE statement. The default for INFORMIX-OnLine is page level locking, and INFORMIX-SE uses only row level locking.

Locking and Performance

Two areas of concern:

- Resource availability
  - rarely infinite
  - limit on the number of locks available
  - will encounter errors if not available
  - if resources are insufficient, performance will suffer
- Concurrency
  - needed to maintain data integrity
  - can place processes in a wait state
  - the lower the concurrency, the poorer the performance

Locking and Performance

There are two areas of concern when considering how locking relates to performance, resource availability and concurrency.

Resource Availability

Lock resources are rarely infinite. No matter how large the limitation may be, there is usually a limit on the number of locks available. In some cases, the limitation is set by:

- the Informix software (64 per table for INFORMIX-SE 5.x).
- the user when the database server is initially set up (INFORMIX-OnLine).

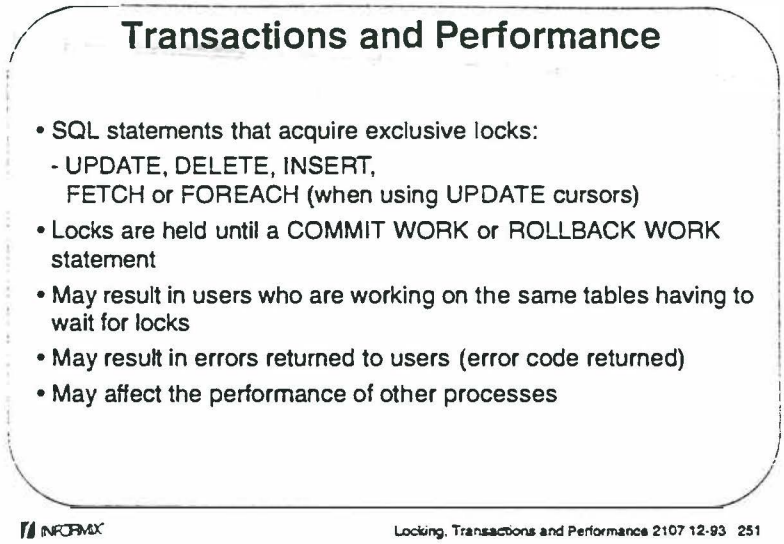
Whenever there is a limitation, there is some concern over the availability of resources. If a database server process requires a lock and none is available, the process will encounter an error. If the allocated lock resources are insufficient for the needs of the application, performance will suffer.

Concurrency

Concurrency refers to the cooperation between two or more processes in accessing and updating data. The granularity level refers to the size of the data object to be locked. The higher the granularity, the lower the concurrency available, and the poorer the performance of the applications are likely to be.

Goal

The goal is to lock the smallest piece of data possible while ensuring data integrity.


Transactions and Performance
• SQL statements that acquire exclusive locks:
- UPDATE, DELETE, INSERT,
FETCH or FOREACH (when using UPDATE cursors)
• Locks are held until a COMMIT WORK or ROLLBACK WORK statement
• May result in users who are working on the same tables having to wait for locks
• May result in errors returned to users (error code returned)
• May affect the performance of other processes
INFORMIX
Locking, Transactions and Performance 2107 12-93 251

Transactions and Performance

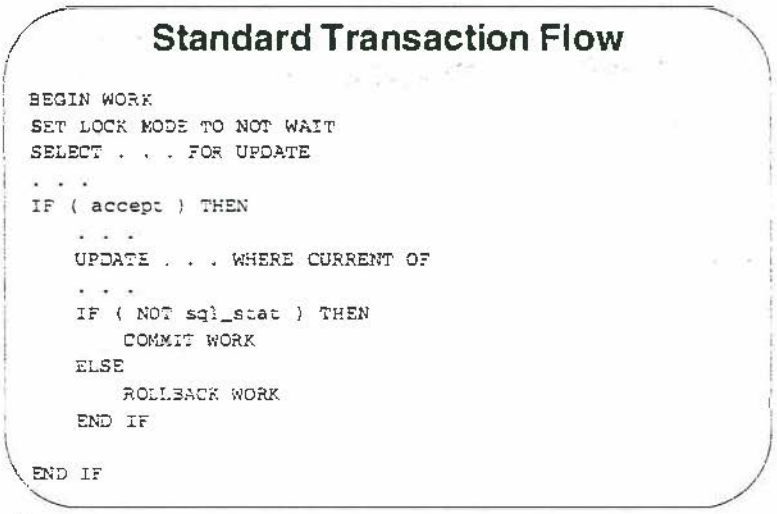
Any UPDATE, DELETE, INSERT, FETCH or FOREACH (when using an UPDATE cursor) operation causes a lock to be held on the rows or pages.

The duration of the locks is as follows for:

- A database without transaction logging - locks are acquired for each singleton SQL statement when it begins and are released when it completes.
- A database with transaction logging - locks are acquired for the programmer-defined transaction when it begins and are released when it completes with a COMMIT WORK or ROLLBACK WORK statement.

A database with transaction logging can hold locks for much longer than a database without transaction logging. The locks placed on the rows or pages are exclusive locks which prevent other users from accessing the same data. If the locks are held for any length of time, concurrency is reduced and performance is affected.

Tip: The length of time locks are held is an important consideration when designing your transaction. If concurrency and performance is a concern, you should consider design alternatives when determining how to implement transactions.



Standard Transaction Flow

The standard transaction flow defines the typical transaction design for an update process. This design is often suitable for maintenance modules that are used infrequently.

The design of this transaction utilizes the INFORMIX exclusive row locking mechanism with implicit adjacent key locking. There are many considerations when using the standard transaction flow:

- The lock duration is dependent upon the time required for the user to perform the data entry.
- Problems arise when another user is attempting to DELETE or INSERT an index row that is adjacent to the exclusive row locked FOR UPDATE.
- A long transaction may occur if a user unintentionally leaves the process in the middle of the data entry.

Adjacent key locking is used in INFORMIX-OnLine Version 5.x. It is a method of B+tree row locking whereby a deleted row is removed from the tree, and an exclusive lock is placed on the row adjacent to the deleted row.

Note: This is not an issue with INFORMIX-OnLine Version 6.0, which uses a new method of locking index keys, called key value locking.

UPDATE Cursors and Transactions

To ensure data integrity, the database server enforces the following rules for UPDATE cursors:

- The fetch must occur within a transaction
- The data must be modified (UPDATE, INSERT, DELETE) within one transaction

Performance concerns:

- Resources are held when it is not necessary
- Length of the transaction may impact resource availability
- Potential scenario for user-initiated long transaction

UPDATE Cursors and Transactions

An UPDATE cursor provides a level of data integrity that is not possible with the UPDATE statement. Since an update lock is placed on the row when it is fetched, you can base your decision to update with new data elements on the original contents of the row.

Disadvantages

Due to the rules enforced by the database server, update cursors lend themselves to:

- holding resources when it is not necessary if the user is primarily viewing the data and has not initiated a decision to update the data.
- large transactions using many resources.
- a long transaction if a user unintentionally leaves the process in the middle of the transaction.

Solution

You can implement alternate designs that will maintain the same degree of data integrity while reducing resources and increasing concurrency, or you can DECLARE an update cursor WITH HOLD and implement smaller transactions (COMMIT WORK at frequent intervals).

Alternate solutions to the standard transaction flow are described in the following pages.

Using UPDATE Cursors WITH HOLD

An UPDATE cursor WITH HOLD can be used to break a long series of updates into smaller transactions with an understanding of the implications:

Cursors without hold:

- Closed by ending the transaction
- Locks are released by ending the transaction

Cursors WITH HOLD:

- Not closed by ending the transaction
- Locks are released by ending the transaction

Using UPDATE Cursors WITH HOLD

There are some program designs that lend themselves to a series of transactions. One scenario is the use of a *master* and *detail* cursor. The *master* cursor is used to scan the table, and selected rows are used to determine if an update is required on a different table. If each UPDATE is treated as a separate transaction, the COMMIT WORK statement following the UPDATE closes all cursors, including the *master* cursor.

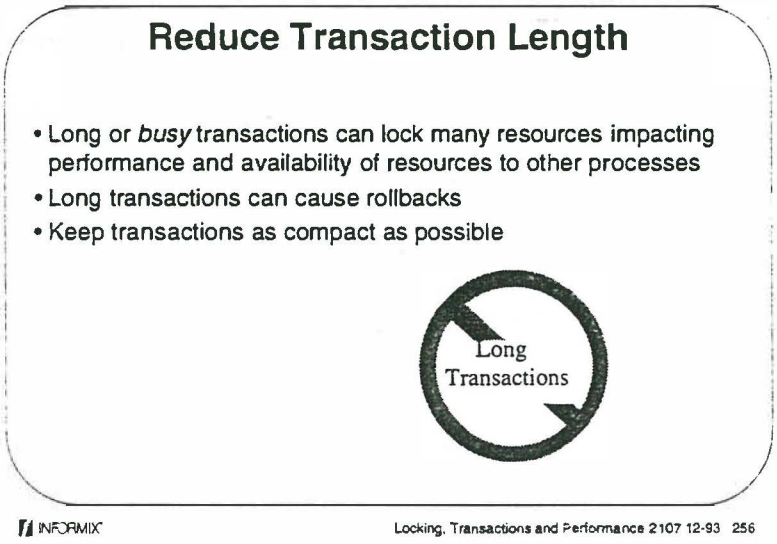
```
DECLARE master_curs CURSOR FOR . . .
DECLARE detail_curs CURSOR FOR . . . FOR UPDATE
OPEN master_curs
WHILE . . .
    FETCH master_curs INTO . . .
    IF ( it is selected ) THEN

        BEGIN WORK
        OPEN detail_curs USING the selected master row
        FETCH detail_curs . . .
        UPDATE . . . WHERE CURRENT OF detail_curs

        IF ( NOT sql_stat ) THEN
            COMMIT WORK
        ELSE
            ROLLBACK WORK
        END IF

    END IF

END WHILE
```

Solution

One solution is to change the transaction design and move the BEGIN WORK statement before the OPEN of the *master* cursor, creating one *large* transaction. However, this solution is impractical if there are a large number of rows to update. A large amount of resources will be required, and concurrency and performance will suffer.

A more practical solution is to declare an update cursor WITH HOLD. A hold cursor is not closed at the end of a transaction. Care must be taken when using update cursors WITH HOLD, understanding the implications that all locks are released at the end of a transaction, including locks placed on the rows fetched with the hold cursor.

```
DECLARE master_curs CURSOR WITH HOLD FOR

DECLARE detail_curs CURSOR WITH HOLD FOR        FOR UPDATE

OPEN master_curs

WHILE . .

    FETCH master_curs INTO . .

    IF ( it is selected ) THEN

        OPEN detail_curs USING the selected master row

        BEGIN WORK

            FETCH detail_curs

            UPDATE . . . WHERE CURRENT OF detail_curs

            IF ( NOT sql_stat ) THEN
                COMMIT WORK
            ELSE
                ROLLBACK WORK
            END IF


END WHILE
```


Reduce Transaction Length
• Long or *busy* transactions can lock many resources impacting performance and availability of resources to other processes
• Long transactions can cause rollbacks
• Keep transactions as compact as possible
Long Transactions
INFORMIX
Locking, Transactions and Performance 2107 12-93 256

Reduce Transaction Length

A transaction is a series of operations (SQL statements) on a database that you want completed as a single unit of work, or not at all. While a transaction is active, it will hold locks for any data that is changed by an UPDATE, DELETE or INSERT statement.

Long Transactions

While locks are held, other processes are denied access and either wait or fail. With INFORMIX-OnLine and a database with transaction logging, it is possible for a process with a very long transaction to fail due to insufficient logical log space. Such a condition is known as a *long transaction* and should be avoided.

In order to maintain the highest degree of concurrency possible and to minimize the likelihood of a long transaction occurring, you should strive to keep transactions as short as possible. Consider reducing the lock duration by re-organizing the transaction.

A transaction should contain the absolute minimum number of operations needed. The shorter and more compact your transactions, the less likely you are to encounter contention or long transaction problems.

Reduce Lock Duration: Lock Indicator

```
SELECT initial_lock_counter, . . .
INPUT # Accept user input
IF ( accept ) THEN
    SET LOCK MODE TO WAIT 10
    BEGIN WORK
    SELECT current_lock_counter, . . . FOR UPDATE
    IF ( initial_lock_counter = current_lock_counter ) THEN
        UPDATE current_lock_counter + 1, . . .
        IF ( NOT sql_stat ) THEN
            COMMIT WORK
        ELSE
            ROLLBACK WORK
        END IF
    ELSE
        # Abort the transaction and display message to user.
```


Reduce Lock Duration: Lock Indicator

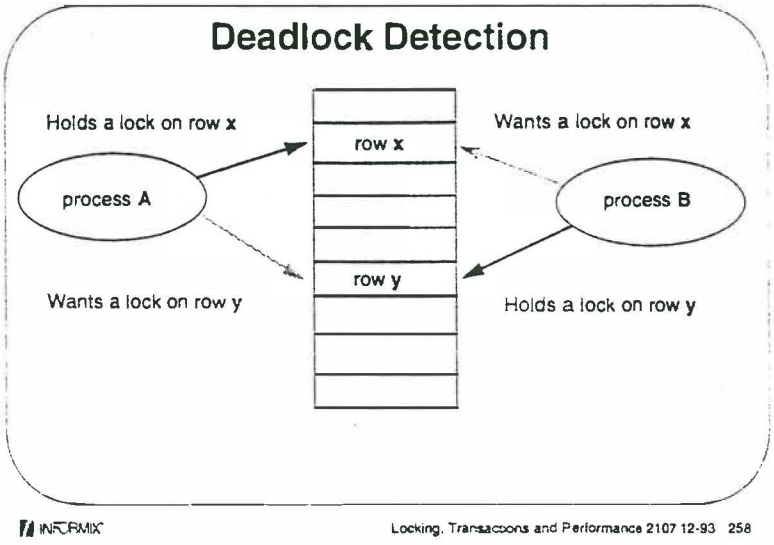
This transaction design improves on the design of the standard transaction flow by reducing the lock duration through the use of a *lock indicator* column on the appropriate table. This design would be used for core modules of an application that requires high throughput and concurrency.

A lock indicator column is added to the appropriate tables. The indicator is used to determine if the row has changed over the course of the user input. The logical steps are as follows:

1. The *before* image of the lock indicator is selected prior to user input.
2. After the user has accepted input, the value of the lock indicator is re-selected FOR UPDATE.
3. The *before* and *after* images are compared.
4. If the *before* and *after* images are identical, increment the value of the lock indicator and complete the transaction.
5. If the *before* and *after* images are different, abort the transaction and display a message to the user.

Advantages and Disadvantages

Implementing an application lock indicator reduces the duration of the exclusive lock. The exclusive lock duration is minimal, and there is greater concurrency for high volume modules. The disadvantage is the user may lose data input and have to re-enter the data. If the table with the lock indicator is very large, the additional space to store the indicator may be an issue.



Deadlock Detection

When multiple processes are accessing the same rows in a table, it is possible for a deadlock to occur.

In the example above, process A holds a lock on row x. It then wants to obtain a lock on a second row, row y. Row y is currently locked by another user, process B. If the lock mode is set to wait, process A will wait for process B to release the lock on row y.

In the meantime, process B is holding a lock on row y, and wants to obtain a lock onrow x. Row x is currently locked by process A. If the lock mode is set to wait, process B will wait for process A to release the lock on row x.

In other words, process A is waiting for process B, and process B is waiting for process A. This is a deadlock situation, as both processes will wait forever. Deadlocks are serious problems, as they can affect performance and halt activity. Informix has a built-in mechanism that detects deadlocks and prevents them from happening.

The database server maintains a list of locks for every user on the system. Before a lock is granted, the lock list for each user is examined. If a lock is currently held on the resource that the process wishes to lock, the owner of that lock is identified, and their lock list is traversed to see if there are lock waits on any locks held by the user wanting the new lock. If there are, the deadlock is detected at that point, and an error message is returned to the user who wanted the lock.

The ISAM error code returned is: **-143 ISAM error: deadlock detected**

Reduce Lock Duration and Deadlock Error

```
SET LOCK MODE TO NOT WAIT
WHILE ( sql_stat = locked AND count < 5 )
    SELECT . . . FOR UPDATE
    LET count = count + 1
END WHILE
IF (sql_stat = locked) THEN
    LET err_msg = "Row is currently locked"
ELSE
    BEGIN WORK
    UPDATE . . .
    IF ( NOT sql_stat ) THEN
        COMMIT WORK
    ELSE
        ROLLBACK WORK
    END IF
```


Reduce Lock Duration and Deadlock Error

The SET LOCK MODE statement controls whether you will wait for a locked row to be freed. You can also specify with the SET LOCK MODE statement how long you wish to wait. Setting the lock mode allows you to wait for that row until the user currently locking the row has released the lock.

While this approach has the advantage of simplicity from the application programming perspective, and it reduces the amount of error checking needed, the process appears to hang while waiting for the lock to be released.

Lock Tries vs. Lock Waits

An alternative is not to wait for locks, and implement a lock retry strategy. If a deadlock occurs as determined by the SQL status, then sleep for a while and try for a pre-determined number of iterations. You retry the transaction at a later time, after the *competing* transaction has had time to be committed or rolled back.

With this approach, you can decide how many times to try to acquire the lock before giving up. An additional benefit is that you can notify the end user that the process is attempting the lock retry.

Appendices

INFORMIX®

Chapter

A

Style Guide for INFORMIX-4GL Development

© 1993 Informix Software, Inc.

Introduction

The purpose of this document is to establish a base set of standards from which to develop applications. While the majority of the style guide is directed around INFORMIX products (INFORMIX-4GL, INFORMIX-OnLine and INFORMIX-SE), the ideas in the pages that follow can be applied to many other development environments.

The ideas in the pages that follow are simply suggestions based on many years of helping customers use INFORMIX products. You may find that some of the suggestions are not in harmony with guidelines that have been in place with previous development efforts.

In those cases, you should evaluate the conflicting issues and determine which of the two should be put forward as the standard for application development. The only right answer is to have a consistent set of guidelines from which to develop application software. It does not matter where the guideline originated, only that once its implementation has been agreed upon, all parties involved adhere to the guideline.

This guideline is separated into five sections as described below:

1.0 Program and Database Standards

This section describes a standard way of writing 4GL programs that enhances readability of the source code. It also describes naming conventions for source files, program variables and database objects.

2.0 Software Development Environment

This section describes the environment under which development of the application should occur. Included in this section is a suggested directory structure, a few development utilities, **Makefiles** and a method of version control.

3.0 Program and Form Design Considerations

This section addresses issues at the design level. Topics include designing for the different INFORMIX servers, national language support (NLS), and user interface issues.

4.0 Attribute Abbreviations

This section suggests attribute abbreviations for software related objects. There are programming related abbreviations that will be used as a standard guideline for naming objects when programming, and general abbreviations for general use.

1.0 Program and Database Standards

1.1 4GL Programming Standards

The following standards will be employed to enhance readability, maintainability, modularity, and define a standard structure within INFORMIX-4GL programs.

1.1.1 Header Block

Include the following header description within every program and globals file:

```
############################################################
#  Module        :  %M%
#  Description   : Holds all error and exception handling routines
#
#  Functions     :  fatal_err
#                   err_recover
#                   err_screen
#                   upd_err_log
#                   get_err_msg
#                   err_msg
#
#  SCCS Id No    :  %W%
#  Author        :  John Smith (INFORMIX SOFTWARE, INC.)
#  Date          :  %H% %T%
#  Path          :  %P%
############################################################
```

Include the following function description prior to every program function and main function:

```
############################################################
#  Function      :  get_err_msg
#  Description   :  selects the appropriate error message from
#                         the error table.
#
#  Incoming Parameters  : err_num - user defined error number.
#  Return Parameters    : <none>
#  Impact Globals       : <none>
#  Perform Screens      : <none>
#
#  Author       Date        Modification Description
#  PTC          11/15/89    Original Code.
#  JLP          03/12/90    Derived from previous err_msg
#                          implementation.
############################################################
```

1.1.2 Globals

All global variables will be declared in a single globals file.

The globals file should be specified through a relative name path. For example:

GLOBALS "../gbl_lib/gbl_main.4gl".

The variable declaration of data types within the GLOBALS/DEFINE section should be grouped by application functionality.

1.1.3 Main, Function, Report

Include the DEFER INTERRUPT and DEFER QUIT statements at the start of each MAIN program. This may be commented out during development and debugging.

Include the WHENEVER ERROR CALL statement at the top of each program function. This may be commented out during development and debugging.

Only one RETURN statement is allowed per FUNCTION.

Within a REPORT function, use the ORDER EXTERNAL BY statement where applicable.

The variable declaration of data types within the MAIN or the FUNCTION DEFINE statement should be grouped and ordered as follows:

ARRAY
RECORD
LIKE
CHAR
DATE
INTEGER
SMALLINT
DECIMAL
MONEY
FLOAT
SMALLFLOAT

1.1.4 Unallowed 4GL Statements

The following INFORMIX-4GL constructs will not be allowed:

GOTO
WHILE <constant>
EXIT WHILE
CONTINUE WHILE
CONTINUE FOR
CONTINUE FOREACH
CASE <variable>
EXIT CASE

1.1.5 Formatting Style

Use descriptive variables and function names, using the underscore (_) to separate words or abbreviations.

The left parenthesis (() and all relational operators must be preceded and followed by a space.

The right parenthesis ()) will be followed and preceded by a space.

Use parentheses () in all conditional statements. For example:

```
WHILE (NOT int_flag AND cnt <= 4 )
```

The comma (,) will be followed by a space.

Capitalize all Informix 4GL and SQL reserved words and 4GL reserved functions.

Use the space character for indentation where each level is indented four spaces.

Do not use tabs to produce the indentation of the code.

The IF-THEN-ELSE statement will follow the format:

```
IF ( <some condition> ) THEN
   < execute some statements >
ELSE
  < execute some other statements >
END IF
```

The WHILE statement will follow the format:

```
WHILE ( <some condition> )
   <execute some statements >

END WHILE
```

The CASE statement will follow the format:

```
CASE
   WHEN ( <some condition> )
      < execute some other statements >
   OTHERWISE
      < execute some other statements >
END CASE
```

The field processing types within the INPUT and CONSTRUCT statements should be grouped and ordered as follows:

```
BEFORE [INPUT | CONSTRUCT]
BEFORE FIELD
AFTER FIELD
ON KEY()
AFTER [INPUT | CONSTRUCT]
```

The field processing types within the INPUT ARRAY statement should be grouped and ordered as follows:

BEFORE INPUT
BEFORE ROW
BEFORE INSERT
BEFORE DELETE
BEFORE FIELD
AFTER FIELD
AFTER INSERT
AFTER DELETE
AFTER ROW
ON KEY()
AFTER INPUT

The report processing types within the REPORT statement should be grouped and ordered as follows:

FIRST PAGE HEADER
PAGE HEADER
PAGE TRAILER
ON EVERY ROW
BEFORE GROUP OF
AFTER GROUP OF
ON LAST ROW

1.2 Form File Standards

The following standards will be employed to enhance readability, maintainability, and to define a standard structure within INFORMIX-4GL form files.

1.2.1 Header Block

Include the following header description within every form file:

```
{#####################################################==
 #  Form          :  %M%
 #  Description   :  defines the generic error handling window.
 #  SCCS Id No    :  %W%
 #  Author        :  John Smith (INFORMIX SOFTWARE, INC.)
 #  Date          :  %H% %T%
 #  Path          :  %P%
#####################################################=}
```

1.2.2 Database Section

Use FORMONLY for the specified database. As a result the tables section will always be omitted.

1.2.3 Screen Section

Indicate the screen dimensions within the SCREEN SIZE section. For example:

SCREEN SIZE 20 BY 80

1.2.4 Attributes Section

Input from a PERFORM screen should be placed into a record variable that has a one-to-one correspondence with a screen record defined in the PERFORM file. Structuring records this way will allow for the INPUT **inp_var.*** FROM **sr_add.*** syntax.

All field tag names within the form should correspond to the program record or program array variable names. This will allow the BY NAME command to be used in DISPLAY and INPUT statements.

For each field tag, the display attributes should be ordered as follows:

AUTONEXT
NOENTRY
REQUIRED
FORMAT
PICTURE
WORDWRAP
COLOR | INVISIBLE
REVERSE
DISPLAY LIKE
DOWNSHIFT | UPSHIFT
DEFAULT
INCLUDE
VALIDATE LIKE
VERIFY
COMMENTS

1.2.5 Instructions Section

The defining of screen records should be prefixed by sr_. For example:

SCREEN RECORD sr_<meaningful name>

The defining of screen record arrays should be prefixed by sa_. For example:

SCREEN RECORD sa_<meaningful name>[<number>]

1.3 Database Interface Standards

The following standards will be employed to ensure efficient and accurate database transaction throughput:

1.3.1 Transactions

All database activity that alters information in the database must occur within a transaction, including singleton transactions.

Set the default ISOLATION level following the opening of the database. Any function that changes the ISOLATION level should change it back to the default.

For a transaction, the BEGIN, COMMIT and ROLLBACK statements should reside within one function.

In instances where database referential integrity does not exist, verify that referential integrity will be maintained before altering information in the database.

If temporary tables are specifically created for any reason, the program needs to direct the table to a pre-determined dbspace rather than accepting the default. Also, the program should specify the initial and next extent sizes.

1.3.2 Error Handling

Use SQLCA.SQLCODE rather than STATUS to determine the status of transactions.

All database statements should be preceded by the WHENEVER ERROR CONTINUE statement and followed by a conditional check of SQLCA.SQLCODE. Following a logical execution of database statements, set WHENEVER ERROR CALL <error function>.

1.3.3 Formatting Style

Reference columns within all SELECT statements by the table and column name.

The following SQL statements should be logically separated onto individual lines as follows:

```
SELECT table_1.column_1, table_1.column_2
FROM table_1
WHERE ( table_1.column_1 = "aaa" )
AND ( table_1.column_2 MATCHES "b*" )
ORDER BY table_1.column_3

DELETE
FROM table_1
WHERE ( table_1.column_1 = "aaa" )
AND ( table_1.column_2 <> "b"  )

INSERT
INTO table_1
VALUES ( gr_aaa.* )

UPDATE table_1
SET table_1.column_1 = "Y"
WHERE ( table_1.column_1 = "aaa" )
```

1.4 Naming Conventions

The following naming conventions will be used for naming database objects, program functions, modules, executables, forms, windows, program variables, screen variables and other software objects.

Unless specified otherwise, names for all of these objects will be assigned using the abbreviations found in the *Attribute Abbreviations* section.

All abbreviations within names will be separated by the underscore character (_), and should not be named in plural form.

1.4.1 Database Naming Conventions

1.4.1.1 Column Naming

The repeated use of a table name within a column name should be avoided, but will not be prohibited. Clarity should be balanced with brevity.

Column names cannot exceed 18 characters.

1.4.1.2 Table Naming

For clarity and readability, if an entity can be fully described by its full name rather than be abbreviated, no abbreviation will be used. For example, all information related to the customer entity will be stored in a table named **customer** rather than **cust**.

Table names cannot exceed 18 characters, of which the first ten characters must define a unique table name.

1.4.1.3 Index Naming

Index naming will follow the convention defined below:

<table abbrev>_<index type>_idx<num> where

<table abbrev> - denotes the first ten characters of the table name. The table abbreviation length should not exceed ten characters.

<index type> - a single letter denoting the type of index as per the following definition:

p	primary key of the table.
s	secondary key(s) on the table.
i	integrity constraint.

<num> - two digit number denoting the order in which the index was created.

The index name cannot exceed 18 characters.

1.4.2 Function, Module, and Executable Names

1.4.2.1 Global File Naming

All global files will begin with the letters **gbl_**, and continue with an abbreviation that represents the program each global file is associated with. Each global file ends with a **.4gl** extension.

Global file names, exclusive of the **.4gl** suffix, should not exceed 12 characters in length. If you are using SCCS, the name should not exceed 10 characters.

1.4.2.2 Function Naming

Function names should be preceded by one of the following abbreviations to reflect a function's primary purpose:

upd	updates database information
ins	inserts or adds information to the database
del	deletes information from the database
sel	retrieves or selects information from the database
dsp	displays information to the user
inp	receives input from the user
err	performs error handling
drv	driver routine for a logical task
rpt	report function
mnu	contains menu logic
scr	contains screen input/output logic

This list is intended only as an initial list. Additional abbreviations will be added as development continues. Functions should be grouped by the tasks they perform, and should be separated into the appropriate module file as defined below.

Function names should not exceed 20 characters in length.

1.4.2.3 Module File Naming

Modules will be named according to the functions they contain. For example, if three functions that perform customer maintenance are contained in a module, that module should begin with the abbreviation ***cust***. If all three functions are related to screen manipulation, the module would be called ***cust_upd***.

Module names should be preceded by one of the following abbreviations to reflect a module's primary purpose:

svc	holds database manipulation routines
scr	holds screen input/output routines
rpt	holds reporting routines
bat	holds batch routines
prg	holds purge routines
lib	holds generic library functions
gbl	holds global variable definitions

All module file names will end with the suffix **.4gl**. Module file names, exclusive of the **.4gl** suffix, should not exceed 12 characters in length. If you are using SCCS, the name should not exceed 10 characters.

1.4.2.4 Executable File Naming

All compiled file names will end with the suffix .4ge. All RDS file names will end with the suffix .4gi. All executable file names, exclusive of the suffix, should not exceed 12 characters in length.

1.4.2.5 Form Naming

A form is any screen that occupies an entire terminal device's display area. No other background forms or portions of other forms should be visible while a form is being displayed. All form names will begin with the characters frm_.

Form names, exclusive of the .per suffix, should not exceed 12 characters in length.

1.4.2.6 Window Naming

A window is any screen portion displayed over an existing form or on top of other windows. All window names will begin with the characters wnd_.

Window names, exclusive of the .per suffix, should not exceed 12 characters in length.

1.4.3 Program Variable and Screen Variable Names

The following prefixes and suffixes will be used in all INFORMIX-4GL programs to describe programming-related objects.

1.4.3.1 Global Variables Prefixes

g_	global simple program variable name defined in a globals 4GL file
ga_	global program array name defined in a globals 4GL file
gr_	global program record name defined in a globals 4GL file

1.4.3.2 Non-Global Variable Prefixes

pa_	non-global program array name defined in a local function
pr_	non-global program record name defined in a local function

1.4.3.3 Screen Variable Prefixes

sa_	screen array name defined in a *.per file
sr_	screen record name from a *.per file

1.4.3.4 Other Software Structure Suffixes

_curs	INFORMIX-4GL cursor names
_pre	statement name used in the PREPARE and EXECUTE statements
_frm	statement name used in the OPEN [WINDOW | FORM] statements

2.0 Software Development Environment

2.1 Directories

Directory names should follow the established naming conventions and should not exceed 15 characters.

Each directory will include a README file. This file should contain a brief description of the purpose of the directory, the files that are allowed in the directory, and any other pertinent information about the directory.

All files should have read only permissions for owner, group and other, and the permissions for each directory should be 775.

A description of the contents of each directory directly below the **devel** directory is listed below:

etc	Contains all development utilities. Developers should include this directory in their path.
lib	Contains compiled library functions.
bin	Contains the modified runner and debugger executables for RDS/ID. Developers should include this directory in their path.
src	Contains all source code for the application. In most cases, another level of directories will exist on the lower level, further segmenting the application source code in logical entities.
obj	Contains a number of sub-directories for the placement of **4go**, **o**, and **frm** files. This directory should be included within the DBPATH.
unl	If unloading of the database needs to be done, the data should be placed in this directory. This directory may also contain load and unload scripts.
make	This directory contains **Makefiles** global to the development environment.
4gi_bin	Contains all **4gi** executables generated through RDS.
4ge_bin	Contains all **4ge** executables generated through **c4gl**.

A description of the contents of each directory directly below the source directory is listed below:

waste	This directory is a place that developers can deposit files that they do not currently need, but feel may be needed in the near future.

	Periodically, the system administrator will remove all files from this directory.
4gl	This directory serves as a repository for every 4GL file in the system. This directory, when used in conjunction with grep, is useful for finding instances of global variables being used, function calls being made, etc.
per	This directory is similar to the 4gl directory and serves as a repository for every per file in the system.
4gl_lib	This directory contains generic 4GL module routines and global files which pertain to all modules within the application.
c_lib	This directory contains all C and ESQL/C source code required by the system, including the C source required for custom runners within RDS/ID.
per_lib	This directory contains .per files that are generic to the entire application.
invoice	This directory is an example of a directory dedicated to a specific module in the application. Each directory will contain the required 4GL, PERFORM, Makefile, and README file.

2.2 Programming Operating Environment Specification

The following environment variables should be defined within each programmer's operating environment:

- Informix-related
  - INFORMIXDIR
  - SQLEXEC
  - DBPATH
  - DBPRINT
  - DBDATE
  - DBTEMP
  - DBEDIT
- Application-related:
  - BASEDIR
  - SCCSDIR

2.3 Development Utilities

During the creation of 4GL program and PERFORM files, the development team should take advantage of several tools that are available.

The **fe** utility should be used to edit 4GL error and program files.

The **pe** utility should be used to edit screen specification form and error files.

The **find_vi** utility searches all **.4gl** source files in a directory for a string specified at the command line.

Other tools will be added on an *as needed* basis. Development team members should feel free to write or make suggestions to the development administrator about tools that would make the development process more productive.

2.4 Makefiles

Makefiles will be used to manage the compilation process. The **Makefiles** will include instructions for generation of both RDS and binary executable files.

Each developer should program using the RDS/ID system, but periodically compile and test a C compiled version of the module. Since the production system will be a C compiled system, each developer needs to ensure that his programs will function properly in that mode.

See the Appendix for sample **Makefiles.**

2.5 Version Control

SCCS version control will be implemented at some point during the project. The decision to implement a version control system will be made by the design team. The procedures and policies regarding version control will be made at the time that a mechanism is put in place.

3.0 Program and Form Design Considerations

3.1 The MAIN Section of a 4GL Program

The MAIN section of each 4GL executable should contain as few instructions as possible. Furthermore, the MAIN section should be placed in a file that contains no other functions. Taking this approach leads to a more flexible set of code for enhancement and support purposes.

There are only a couple of instructions (DEFER QUIT and INTERRUPT) that are required to be placed in the MAIN section.

Below is a sample MAIN section from a program called **orders.4ge**.

```
DATABASE orders      #opens the database, optional statement

MAIN

    WHENEVER ERROR <error handling>
    DEFER INTERRUPT #required to be in MAIN

    CALL orders_main()  #controlling function for orders

END MAIN
```

3.2 Input Features, Options, and Standard Practices

The INPUT WRAP feature should be active for most input screens. This feature causes the cursor to return to the first position of the first input field after the user presses the return key on the last input field. The end result of setting this option is that the user must terminate the input session by pressing some key other than **return**.

This option should be set in a function that sets the other Informix options at the beginning of the program. Some input screens may require this feature to be disabled. If so, the function should set INPUT NO WRAP at the beginning of the function and set the value back to WRAP before exiting the function.

Input field lengths will be indicated to the user through default delimiters in the form specification files. These delimiters are the bracket characters ([]). Reverse video and color will not be used to indicate input field lengths.

The INT_FLAG variable should be set to FALSE immediately prior to executing any of the following statements:

- INPUT
- INPUT ARRAY
- DISPLAY ARRAY
- PROMPT
- CONSTRUCT

Example:

```
IF INT_FLAG = TRUE THEN
    LET INT_FLAG = FALSE
INPUT BY NAME < variable name >
```

3.3 Using Function Keys

INFORMIX-4GL provides the ability to detect the pressing of function keys while waiting for the user input. While this provides a convenient mechanism for the user, care should be taken when writing programs that take action based on function key detection. This is due to the way that function keys are implemented under UNIX as well as the fact that not all terminal input devices have the same number of function keys.

When writing a program that takes action based on function keys, try to provide an alternative control key character that takes the same action. Taking this precautionary step will protect the user when:

1. The requested function key is not available on the terminal input device.
2. The pressed function key is being intercepted and interpreted as an instruction to a layer of software that is running below the 4GL program. This is common when a PC is running a DOS terminal emulation package that provides connection to the UNIX machine.
3. The escape sequence being sent when the function key is pressed is broken into pieces and sent separately, causing a time-out condition to occur. The function key is then ignored. This can occur when a network is completely saturated or in cases where the escape sequence travels a long distance over a packet switching network.

3.4 Using Control Keys

In addition to being able to detect function keys, INFORMIX-4GL has the ability to detect and take actions based on the user pressing control keys. Of the control keys, **A-Z**, some are reserved by the operating system and some are reserved by Informix. This section lists each control key and states if it is available for use in 4GL programming.

Control Key	Notes
A	Reserved by Informix for field editing. Toggles between insert and type over mode.
B	Available for 4GL
C	Available but not recommended. This key is commonly used for canceling user input from within a 4GL program. It is also commonly used in UNIX to cancel process execution.
D	Reserved by Informix for field editing. Deletes from the cursor position to the end of the field.
E	Available for 4GL

F	Available for 4GL
G	Available for 4GL
H	Reserved by Informix for field editing. Moves the cursor non-destructively one space to the left.
I	Available, but not recommended. The ASCII sequence for TAB and Control-I are equivalent. Thus using this key in 4GL could interfere with normal data entry procedures.
J	Available but not recommended. May interfere with normal data entry procedures.
K	Available but not recommended. May interfere with normal data entry procedures.
L	Reserved by Informix for field editing. Moves the cursor non-destructively one space to the right.
M	Available but not recommended. The ASCII sequence for ENTER or RETURN and Control-M are equivalent. Using this key in 4GL could interfere with normal data entry procedures.
N	Available for 4GL
O	Available for 4GL
P	Available for 4GL
Q	Reserved by the operating system for flow control.
R	Reserved by Informix to refresh the screen.
S	Reserved by the operating system for flow control.
T	Available for 4GL
U	Available for 4GL
V	Available for 4GL
W	Available for 4GL. Used as a help key by Informix products.
X	Reserved by Informix for field editing. Deletes the character beneath the cursor.
Y	Reserved by some operating systems for process control.
Z	Reserved by some operating systems for process control.

3.5 Required Key Assignments

The following keys should be defined for every input situation. In order to provide a consistent look and feel to the user, it is highly desirable to have these keys identical for every input situation for every application.

Key Name	Description/Default/Recommendation
ACCEPT	User presses this key to indicate completion of data entry. It is set through the OPTIONS command. Default: ESC (**Escape**) Recommendation: Use default. Using this key does impose a slight delay in processing due to Informix's use of the curses libraries.
INTERRUPT	User presses this key to interrupt or abort the current input situation (input, menu, etc.). It is set at the operating system level through the **stty** utility. Default: Depends on the operating system, but usually **Control-C**. Recommendation: Use **Control-C**
INSERT	Used to insert a row into an input array. Default: **F1** Recommendation: Use default
DELETE	Used to delete a row from an input array. Default: **F2** Recommendation: Use default
NEXT	Used to page down in input and display arrays. Default: **F3** Recommendation: Use default
PREVIOUS	Used to page up in input and display arrays. Default: **F4** Recommendation: Use default
HELP	Used to initiate help. Default: **Control-W** Recommendation: Use **F10**. This presumes that 10 function keys will be available, but if not, only the help feature is affected.

3.6 Reserved Lines

INFORMIX-4GL uses several lines for the displaying of messages, errors, prompts, and other optional information. The vertical positioning of these lines can be determined by the programmer during application development. Program instructions direct INFORMIX-4GL to display these messages at particular offsets within the current window or form.

Do not reference absolute line numbers. Use relative numbers with the LAST and FIRST keywords.

Below is a list of lines that must be defined. With each line is a description of when it is used, the default value, and a recommended value for all applications.

MESSAGE LINE	Determines on which line inside a window all messages will appear. Messages appear on the screen by using the MESSAGE command. Default: FIRST + 1 (line 2) Recommendation: Use default
PROMPT LINE	Determines on which line inside a window all prompts will appear. Prompts appear as the result of using the PROMPT command. Default: FIRST (line 1) Recommendation: Use default
MENU LINE	Determines on which line inside a window all menus will appear. Default: FIRST (line 1) Recommendation: Use default
COMMENT LINE	Determines which line in a window comments will appear. Comments are a feature of form specification files. If implemented for a particular field, then any time the cursor is in the field, the associated comments will appear on the specified comment line. Default: LAST Recommendation: Use default
ERROR LINE	Determines which line on the screen errors will appear. Errors appear on the screen as the result of using the ERROR command, or they can be generated by Informix if the user enters invalid data that is trapped. Default: LAST

	Recommendation: Use default
FORM LINE	Determine on which line of a window the first line of a form will be displayed. Default: FIRST + 2 (line 3) Recommendation: Use default for full screen windows. This allows a ring menu to occupy the first two lines of the window. For smaller windows that are opened within a full screen window, displaying the first line of the form in the first line of the window is usually desirable. This can be accomplished through the ATTRIBUTE section of the OPEN WINDOW statement, allowing the default value to be left unaltered for all other windows.

3.7 Using DISPLAY AT with Perform Screens

As much as possible, the displaying of information to the screen should be driven through the form specification file. That is, fields or constant values should be specified in the form specification file rather than executing the DISPLAY AT command for writing to the screen.

Taking this approach leads to easier screen management. However, there may be times when DISPLAY AT is appropriate. In order to avoid situations where the form specification file conflicts with DISPLAY AT commands, the following guidelines should be used when using DISPLAY AT with PERFORM screens.

1. Assuming the FORM LINE is set to 3, using DISPLAY AT with rows one or two is safe and does not conflict with the form.
2. By convention, do not place any fields in the first two lines of the form. Doing this allows DISPLAY AT to use lines three or four.
3. If room is needed at the bottom of the screen for DISPLAY AT, do not place any fields on the last three lines of the form. Allowing for COMMENTS on the last line of the form, this will allow DISPLAY AT for the two lines above the last line of the form.

3.8 Guidelines for Form Layout

This section will suggest some guidelines for form specification files that will make up the screens of an application. The suggestions below are by no means requirements, and modifications of the items to meet certain user requirements is welcome.

3.8.1 Full Screen Forms

Full screen forms should be opened within a window without a border. Since the form will occupy the entire screen, there is no reason to border the form. Since there is likely to be a ring menu above the form, place a dashed line as the first line in the SCREEN section of the form specification file. This dashed line will provide a visual break

between the menu and form. If space allows, leave the first two lines below the dashed line blank. This provides a visual break between the menu and form. Three pieces of information should be written to this line through the 4GL program. The information will overlay the dashed characters. The pieces of information are as follows:

left justified	The date the screen was entered by the user.
center	The title of the screen.
right justified	The current page and total number of pages for a multi-page data entry screen. This will only appear on the screen if the data entry screen contains more than one page.

3.8.2 Partial Screen Forms

There is often a need to open a small window to prompt the user for a yes/no answer, or open a small window showing valid data elements for the data entry being done on a full screen form. Forms of this type should be opened with a border, giving a visual break from the screen that lies behind the pop-up window.

4.0 Suggested Attribute Abbreviations

The following attribute abbreviations will be used to name all software related structures. Included in this list are two types of abbreviations: programming-related and general.

Programming-related abbreviations are those that will be of primary interest to programmers: *sa* for screen array, *pa* for program array, etc. General abbreviations are included for general use: *mon* for Monday, *jan* for January.

abbreviated	abbrv
account	acct
action	actn
address	addr
agency	agcy
amount	amt
annual	anl
April	apr
area code	ac
array	arr
August	aug
average	avg
beginning, begin	beg
boolean	bool
business	busn
change	chg
clear	clr
code	cd
comment	cmt
company	co
continent	cont
contact	cntc
country	ctry
count	cnt
currency	curr
current	crnt

cursor	curs
database	db
daily	dly
date	dt
December	dec
decision	decsn
default	def
delete	del
department	dept
description	dsc
destination	dest
difference	diff
directory	dir
display	dsp
domestic	dom
double	dbl
effective	eff
employee	emp
error	err
estimate	est
extension	ext
February	feb
first	frst
flag	flg
frequent	freq
Friday	fri
general	gen
globals	gbl
history	hst
holiday	hldy
hold	hld
ignore	ign
indicator	ind

identification	id
index	idx
individual	indv
insert	ins
international	intl
information	info
inventory	inv
January	jan
July	jul
Julian	juln
June	jun
last	lst
length	len
line	ln
location	loc
long	lg
March	mar
manager	mgr
market	mkt
maximum	mx
May	may
meeting	mtg
member	memb
message	msg
middle initial	mi
minimum	mn
miscellaneous	misc
monday	mon
month	mth
multiple	mult
name	nam
negative	neg
node	nd

November	nov
number	num
October	oct
organization	org
paid	pd
pattern	ptrn
page	pg
period	per
phone	ph
policy	plcy
positive	pos
program array	pa
profile	prfl
property	prop
quota	quo
range	rng
rank	rnk
rate	rt
record	rec
region	regn
request	req
return	ret
remark	rmk
return	rtn
revenue	rev
sales	sls
Saturday	sat
screen array	sa
screen record	sr
scope	sc
select	sel
September	sep
service	svc

short	sh
start	strt
string	str
source	src
space	spc
Sunday	sun
single	sgl
state	st
status	stat
Thursday	thu
time	tm
total	tot
type	typ
Tuesday	tue
update	upd
user	usr
value	val
week	wk
Wednesday	wed
window	wnd
year	yr
zip code	zip

INFORMIX®

Chapter

B

Sample Makefiles

© 1993 Informix Software, Inc.

Appendix: Sample Makefiles

```
#########################################################################
# Makefile          : %M%
#
# Description: Creates <module name> module and forms.
#
# SCCS Id No        : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#########################################################################

include ${PROJECT}/make/incl_all.com
include ${PROJECT}/make/incl_all.def

PROG_4GE    = <prog_nam>.4ge
PROG_4GI    = <prog_nam>.4gi

LIB_4GE     = $(PROJECT_LIB_4GE)
LIB_4GI     = $(PROJECT_LIB_4GI)

GBL_4GL     = <file_nam>_g.4gl

SOURCE      = <file_nam>_g.4gl <file_nam>_q.4gl <file_nam>_a.4gl
\
                <file_nam>_u.4gl <file_nam>_s.4gl
<fil_nam>_s.4gl

SOUB_SRC = stub.4gl

PER = <form_nam>_if.per <form_nam>_qf.per

STUB_PER =

include ${PROJECT}/make/incl_all.dep
```

```
#####################################################################
# Makefile          : %M%
#
# Description: Creates the <library_nam> library routine.
#
# SCCS Id No        : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#####################################################################

include ${PROJECT}/make/incl_all.com
include ${PROJECT}/make/incl_all.def

LIB_4GE     = $(<APPL>_<LIB_NAM>_LIB_4GE)

GBL_4GL     = <lib_nam>_gbl_1.4gl

SOURCE      = <lib_nam>_gbl_1.4gl <lib_nam>_msc_1.4gl
                <lib_nam>_sql_1.4gl

include ${PROJECT}/make/incl_lib.dep
```

```
#####################################################################
# Makefile          : %M%
#
# Description: Defines the compilation rules to be included
#                       within all makefiles.
#
# SCCS Id No        : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#####################################################################

.SUFFIXES: .o .ec .c .4gl .4go .4gi .per .frm .msg .iem

C_OPT = -I${INFORMIXDIR}/incl -c

# ----------------- #
# Compilation Rules #
# ----------------- #

.4gl.4go:
                fglpc $*.4gl

.4gl.o:
                c4gl -c $*.4gl
                @- rm $*.c $*.ec

.ec.o:
                -esql -c $*.ec
                @- rm $*.c

.c.o:
                cc ${CFLAGS} ${C_OPT} $*.c

.per.frm:
                form4gl -s $*.per

.msg.iem:
                mkmessage -s $*.msg $*.iem
```

```
#####################################################################
# Makefile          : %M%
#
# Description:          Defines applicable pathnames, filenames,
#                       and libraries for all makefiles.
# SCCS Id No        : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#####################################################################

#-------------------------------#
# Development Environment Paths #
#-------------------------------#

LIB_PATH = ${BASEDIR}/lib/obj
BIN_PATH = ${BASEDIR}/${APPLDIR}/bin
FRM_PATH = ${BASEDIR}/${APPLDIR}/frm
UTIL_PATH = ${DEV_HOME}/etc

#----------------------------#
# Training Environment Paths #
#----------------------------#

TRN_ENV = test
TRN_BASEDIR = prj/useb

TRN_PATH = $(TRN_ENV):/$(TRN_BASEDIR)/${APPLDIR}
TRN_FRM = $(TRN_PATH)/frm
TRN_BIN = $(TRN_PATH)/bin

#--------------------------------#
# Unit Testing Environment Paths #
#--------------------------------#

TST_ENV = test
TST_BASEDIR = tst/susp

TST_PATH = $(TST_ENV):/$(TST_BASEDIR)/${APPLDIR}
TST_FRM = $(TST_PATH)/frm
TST_BIN = $(TST_PATH)/bin
```

```
#--------------------------------------#
# Integration Testing Environment Paths #
#--------------------------------------#

INT_TST_ENV = resgg
INT_TST_BASEDIR = prj

INT_TST_PATH = $(INT_TST_ENV):/$(INT_TST_BASEDIR)/$(APPLDIR)
INT_TST_FRM = $(INT_TST_PATH)/frm
INT_TST_BIN = $(INT_TST_PATH)/bin
#-------#
# Files #
#-------#

SCHEMA = $(BASEDIR)/$(APPLDIR)/schema.sql

GBL_4GL = $(BASEDIR)/lib/4gl/prj_gbl_1.4gl

RUNNER = prjgo
DEBUGGER = prjdb

TAGS = tags
SET_EXPLAIN = sqexplain.out
NOHUP = nohup.out
DIFFS = table_diffs index_diffs

TST_MAIL = login_name

OTH_CLEAN = $(TAGS) $(SET_EXPLAIN) $(NOHUP) $(DIFFS)

#---------------------#
# Generic 'C' Library #
#---------------------#

PROJECT_LIB_C = $(LIB_PATH)/prj_clib.a

#-----------------------------#
# Standard PROJECT Libraries #
#-----------------------------#

STD_PROJECT_LIB_4GI = $(LIB_PATH)/prj_4lib.4ga
STD_PROJECT_LIB_4GE = $(STD_PROJECT_LIB_4GI:.4ga=.a)

STD_LIB_4GI = $(STD_PROJECT_LIB_4GI)
STD_LIB_4GE = $(STD_LIB_4GI:.4ga=.a) \
                    $(PROJECT_LIB_C)
```

```
#############################################################
# Makefile          : %M%
#
# Description:          Defines the dependencies to be included
#                       within all 4gl makefiles.
#
# SCCS Id No : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#############################################################

#--------#
# Macros #
#--------#

FRM         = $(PER:.per=.frm)
OBJ_4GE     = $(SOURCE:.4gl=.o)
OBJ_4GO     = $(SOURCE:.4gl=.4go)

STUB_FRM            = $(STUB_PER:.per=.frm)
STUB_OBJ_4GE        = $(STUB_SRC:.4gl=.o)
STUB_OBJ_4GO        = $(STUB_SRC:.4gl=.4go)

_PROG_4GE = $(BIN_PATH)/$(PROG_4GE)

#--------------#
# Dependencies #
#--------------#

all:
    @ echo "Usage: make
    [4gi|4ge|form|test|clean|xclean|<filename>]"

4gi:    form $(PROG_4GI)
        @ echo "Linking: "$(PROG_4GI)
        @ cat $(STUB_OBJ_4GO) $(OBJ_4GO) $(LIB_4GI) > $(PROG_4GI)

4ge:    form $(_PROG_4GE)
        @ echo "Linking: "$(_PROG_4GE)
        @ c4gl -O -o $(_PROG_4GE) \
        `lorder $(STUB_OBJ_4GE) $(OBJ_4GE) | tsort 2> /dev/null` \
        $(LIB_4GE)

        @ strip $(_PROG_4GE)
        @ chmod 775 $(_PROG_4GE)
```

```
form:   $(FRM) $(STUB_FRM)
        @-cp $(FRM) $(FRM_PATH) > /dev/null 2>&1

clean:
    @-rm -f $(STUB_OBJ_4GO) $(STUB_OBJ_4GE) $(OBJ_4GO) $(OBJ_4GE)
    @ echo "Finished Cleaning Object Files."

xclean:     clean
            @-rm -f $(FRM) $(PROG_4GI) $(_PROG_4GE) $(OTH_CLEAN)
            @ echo "Finished Cleaning Program Files."

$(OBJ_4GE): $(GBL_4GL)
$(OBJ_4GO): $(GBL_4GL)

$(_PROG_4GE): $(OBJ_4GE) $(STUB_OBJ_4GE)
$(PROG_4GI): $(OBJ_4GO) $(STUB_OBJ_4GO)
```

```
#################################################################
# Makefile          : %M%
#
# Description : Defines the dependencies to be included with
#                       modules that require '4ge' compile
#                       execution only and need both 'C' and
#                       '4gl' files.
#
# SCCS Id No        : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#################################################################

#--------#
# Macros #
#--------#

FRM = $(PER:.per=.frm)

OBJ_C = $(C_SOURCE:.c=.o)

OBJ_4GE = $(SOURCE:.4gl=.o)

STUB_OBJ_4GE = $(STUB_SRC:.4gl=.o)

_PROG_4GE = $(BIN_PATH)/$(PROG_4GE)

#--------------#
# Dependencies #
#--------------#

all:
    @ echo "Usage: make
    [4ge|form|test|train|integrate|clean|xclean|<filename>]"

4ge:    form $(OBJ_C) $(_PROG_4GE)
        @ echo "Linking: "$(_PROG_4GE)
        @ c4gl -O -o $(_PROG_4GE) \
            `lorder $(STUB_OBJ_4GE) $(OBJ_4GE) $(OBJ_C) | \
            tsort 2> /dev/null` -ls $(LIB_4GE)

        @ strip $(_PROG_4GE)
        @ chmod 775 $(_PROG_4GE)
```

```
form:   $(FRM) $(STUB_FRM)
        @-cp $(FRM) $(FRM_PATH) > /dev/null 2>&1

test:
        @ echo "Copying Executables to: $(TST_BIN)."
        @-rcp $(_PROG_4GE) $(TST_BIN) > /dev/null 2>&1

        @ echo "Copying Forms to: $(TST_FRM)."
        @-rcp $(FRM) $(TST_FRM) > /dev/null 2>&1

        @ echo "$(TST_BIN)/$(_PROG_4GE)\n\n$(TST_FRM)/$(FRM)\n"
\
            mailx -s "$(APPLDIR): $(_PROG_4GE)" $(TST_MAIL)

integrate:
        @ echo "Copying Executables to: $(INT_TST_BIN)."
        @-rcp $(_PROG_4GE) $(INT_TST_BIN) > /dev/null 2>&1

        @ echo "Copying Forms to: $(INT_TST_FRM)."
        @-rcp $(FRM) $(INT_TST_FRM) > /dev/null 2>&1

train:
        @ echo "Copying Executables to: $(TRN_BIN)."
        @-rcp $(_PROG_4GE) $(TRN_BIN) > /dev/null 2>&1

        @ echo "Copying Forms to: $(TRN_FRM)."
        @-rcp $(FRM) $(TRN_FRM) > /dev/null 2>&1

clean:
        @-rm -f $(STUB_OBJ_4GE) $(OBJ_4GE) $(OBJ_C)
        @ echo "Finished Cleaning Object Files."

xclean:     clean
            @-rm -f $(FRM) $(_PROG_4GE) $(OTH_CLEAN)
            @ echo "Finished Cleaning Program Files."

$(_PROG_4GE): $(OBJ_4GE) $(OBJ_C) $(STUB_OBJ_4GE)
```

```
#####################################################################
# Makefile          : %M%
#
# Description: Defines the dependencies to be included within the
#                       'C' library makefiles.
#
# SCCS Id No : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#####################################################################

#--------#
# Macros #
#--------#

OBJ_C = $(SOURCE:.c=.o)

#--------------#
# Dependencies #
#--------------#

all:        @ echo "Usage: make
            [lib|runner|debug|clean|xclean|<filename>]"

lib:        $(LIB_C)
            @ echo "Making Library : "$(LIB_C)
            @ ar rc $(LIB_C) `lorder $(OBJ_C) | tsort 2> /dev/
null`

runner:     $(OBJ_C)
            @ echo "Making Custom Runner : "$(RUNNER)
            @ cfglgo fgiusr.c $(OBJ_C) -o $(RUNNER)

            @ mv $(RUNNER) $(UTIL_PATH)
            @ chmod 775 $(UTIL_PATH)/$(RUNNER)

debug:      $(OBJ_C)
            @ echo "Making Customer Debugger : "$(DEBUGGER)
            @ cfgldb fgiusr.c $(OBJ_C) -o $(DEBUGGER)

            @ mv $(DEBUGGER) $(UTIL_PATH)
            @ chmod 775 $(UTIL_PATH)/$(DEBUGGER)
```

```
clean:
            @-rm -f $(OBJ_C)
            @ echo "Finished Cleaning Object Files."

xclean:     clean
            @-rm -f $(LIB_C) $(UTIL_PATH)/$(DEBUGGER) \
                        $(UTIL_PATH)/$(RUNNER) $(OTH_CLEAN)
            @ echo "Finished Cleaning Program Files."

$(LIB_C): $(OBJ_C)
```

```
#################################################################
# Makefile          : %M%
#
# Description: Defines the dependencies to be included within
#                       the form only makefiles.
# SCCS Id No        : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#################################################################

#--------------#
# Dependencies #
#--------------#

all:
        @ echo "Usage: make
        [form|test|train|integrate|clean|xclean|<filename>]"

form: $(FRM) $(STUB_FRM)
        @-cp $(FRM) $(FRM_PATH) > /dev/null 2>&1

test:
        @ echo "Copying Forms to: $(TST_FRM)."
        @-rcp $(FRM) $(TST_FRM) > /dev/null 2>&1

        @ echo "$(TST_FRM)/$(FRM)\n" | \
                    mailx -s "$(APPLDIR): forms" $(TST_MAIL)

integrate:
        @ echo "Copying Forms to: $(INT_TST_FRM)."
        @-rcp $(FRM) $(INT_TST_FRM) > /dev/null 2>&1

train:
        @ echo "Copying Forms to: $(TRN_FRM)."
        @-rcp $(FRM) $(TRN_FRM) > /dev/null 2>&1

clean:
        @-rm -f $(FRM) $(STUB_FRM)
        @ echo "Finished Cleaning Object Files."

xclean:     clean
            @-rm -f $(OTH_CLEAN)
```

```
#################################################################
# Makefile          : %M%
#
# Description    : Defines the dependencies to be included within
#                       the library makefiles.
# SCCS Id No        : %W%
# Path              : %P%
#
# Author            : <full_nam>
# Date              : %H% %T%
#################################################################
#--------#
# Macros #
#--------#
LIB_4GI = $(LIB_4GE:.a=.4ga)

OBJ_4GE = $(SOURCE:.4gl=.o)
OBJ_4GO = $(SOURCE:.4gl=.4go)

#--------------#
# Dependencies #
#--------------#
all:
        @ echo "Usage: make [4gi|4ge|clean|xclean|<filename>]"

4gi:    $(LIB_4GI)
        @ echo "Making Library : "$(LIB_4GI)
        @ cat $(OBJ_4GO) > $(LIB_4GI)

4ge:    $(LIB_4GE)
        @ echo "Making Library : "$(LIB_4GE)
        @ ar rc $(LIB_4GE) `lorder $(OBJ_4GE) | tsort 2> /dev/
null`

clean:
        @-rm -f $(OBJ_4GO) $(OBJ_4GE)
        @ echo "Finished Cleaning Object Files."

xclean:     clean
            @-rm -f $(FRM) $(LIB_4GI) $(LIB_4GE) $(OTH_CLEAN)
            @ echo "Finished Cleaning Program Files."

$(OBJ_4GE): $(GBL_4GL)
$(OBJ_4GO): $(GBL_4GL)

$(LIB_4GE): $(OBJ_4GE)
$(LIB_4GI): $(OBJ_4GO)
```

INFORMIX®

Chapter

C

4GL Utilities Source Code

© 1993 Informix Software, Inc.

Appendix: fe

The fe Utility Script

```
#################################################################
# Script        : %M%
# Synopsis      : fe <filename>
# Description   : fe allows editing of the ".err" file resulting
#                 from an unsuccessful compilation of an INFORMIX-
#                 4GL source file. The results of the editing are
#                 saved in the corresponding source file ".4gl"
# Diagnostics   : The exit status is 0 on normal termination and
#                 1 on errors
# SCCS Id No    : %W%
# Path          : %P%
# Date          : %H% %T%
# Modification  : %G% %U%
# Author        : <name>
#################################################################

# Definitions

PROG='basename $0'

SRCFILE=$1".4gl"
ERRFILE=$1".err"

#Check Usage

if [ $# -ne 1 ]
then
    echo "usage: $PROG <filename>" >&2
    exit 1
fi

#Check existence and integrity of file

if [ ! -s $ERRFILE ]
then
    echo "$PROG: \"$ERRFILE\" does not exist. " >&2
    exit 1
fi

if grep -v "^!" $ERRFILE | diff - $SRCFILE > /dev/null 2>&1
then
    vi +/"^!.See.error" $ERRFILE

# Verify that the error file still exists prior to making changes
```

```
    if [ ! -s $ERRFILE ]
    then
        echo "$PROG: \"$ERRFILE\" no longer exists. No changes
                saved. " >&2
        exit 1
    fi

#If the error file exists, then save changes and remove the error
#lines

    sed '/^|/d' $ERRFILE > $SRCFILE
    rm $ERRFILE
    exit 0
else
    echo "$PROG: ERROR: \"$ERRFILE\" not consistent with
          \"$SRCFILE\"" >&2
    exit 1
fi
```

Appendix: pe

The pe Utility Script

```
#####################################################################
# Script         : %M%
# Synopsis       : pe <filename>
# Description    : pe allows editing of the ".err" file resulting
#                  from an unsuccessful compilation of an INFORMIX-
#                  4GL perform file. The results of the editing are
#                  saved in the corresponding file ".per"
# Diagnostics    : The exit status is 0 on normal termination and
#                  1 on errors
# SCCS Id No     : %W%
# Path           : %P%
# Date           : %H% %T%
# Modification   : %G% %U%
# Author         : <name>
#####################################################################

#Definitions

PROG='basename $0'

PERFILE=$1".per"
ERRFILE=$1".err"

#Check usage

if [ $# -ne 1 ]
then
    echo "usage : $PROG <filename> >&2
    exit 1
fi

#Check existence of file

if [ ! -s $ERRFILE ]
then
    echo "$PROG: \"$ERRFILE\" does not exist." >&2
    exit 1
else
    vi +/"#.See.error" $ERRFILE

    if [ ! -s $ERRFILE ]
    then
    echo "$PROG: \"$ERRFILE\"no longer exists. No changes saved."
            >&2
        exit 1
    fi
```

```
    sed '/^#d' SERRFILE > SPERFILE
    rm SERRFILE
    exit 0
fi
```

Appendix: find_vi

The find_vi Utility Script

```
#################################################################
# Script        : %M%
# Synopsis      : find_vi <search string>
# Description   : Searches all .4gl files in a directory for a
#                 string specified at the command line. For each
#                 file found containing the string, the program
#                 will start "vi" on the file at the point of the
#                 search string location.
# Diagnostics   : The exit status is 0 on normal termination and
#                 1 on errors
# SCCS Id No    : %W%
# Path          : %P%
# Date          : %H% %T%
# Modification  : %G% %U%
# Author        : <name>
#################################################################

# Check there is a command line argument

if [ $# -lt 1 ]
then
    echo "Syntax: $0 <search string>"
    exit 1
fi

GREP_ARG="'$*'"
VI_ARG=$*

#Loop through all 4gl files in the current directory

for i in `ls ./*.4gl`
do
# Run grep and save it's exit status.0 indicates a match was found

    eval grep -i -q $GREP_ARG $i; XSTAT=$?

# If the exit status of grep is 0, then run vi on that file

    if [ $XSTAT -eq 0 ]
    then
        vi +/"$VI_ARG" $i
    fi
done
```

Appendix: 4gltags

The 4gltags Utility

```
#############################################################
# Script     :   4gltags
#
# Synopsis   :   4gltags [-i] [-o] <directory names(s)>
#
# Options    :   -i include standard PROJECT library tags
#                -o include "tags" file(s) located in the
#                following directory(s)
#
# Description : Generates an Informix 'tags' file for FUNCTION
#               and REPORT within all '.4gl' files. Assumes the
#               reserved words 'function' and 'report' are
#               capitalized.
#
#               If "-i" is on the command line, the script
#               includes the "tags" file located in $BASEDIR/lib
#               /4gl.
#
#               If "-o" is on the command line, the script assumes
#               that all following arguments are directory names
#               where files named "tags" exist. Do not include the
#               "tags" filename in the directory path.
#
# Installation : Create a '.exrc' file in your home directory and
#                place the following lines within the file:
#
#                           set tags=./tags
#                           map ] ^]
#                           map [ ^^
#
#               The right bracket (']') character initiate the
#               viewing of the called function when the cursor
#               is placed on the function name following the CALL.
#
#               The left bracket (']') character returns viewing
#               to the calling function.
#
# Diagnostics : The exit status is 0 on normal termination.
#
# SCCS Id No : %W%
# Path       : %P%
# Author     : <full_nam>
# Date       : %H% %T%
#############################################################
```

```
TAGS_FILE="./tags"
LOCAL_PATH=`pwd`
TMP_FILE=$$.tmp

# List of library directories.

LIB_TAGS="$BASEDIR/lib/4gl $BASEDIR/lib/pull_down $BASEDIR/lib/
rslt_lst "

# Remove existing tags file.

if [ -r "$TAGS_FILE" ]
then
 rm $TAGS_FILE
fi

# Create the local tags file.

awk '
/^FUNCTION/{ split( $2, a, "(" );
printf( "%s\t%s\t/^%s %s\n", a[1], FILENAME, $1, a[1] ) >>
    (ofile); }

/^REPORT/{ split( $2, a, "(" );

printf( "%s\t%s\t/^%s %s\n", a[1], FILENAME, $1, a[1] ) >>
    (ofile); }

' ofile=$TAGS_FILE $LOCAL_PATH/*.4gl

# Move the tags file to a working file.

mv $TAGS_FILE $TMP_FILE

# Process the command line arguments.

others=0
for i in $*
do
    case $i in

# Append the tags files for each of the libraries.

        -i)
                for t in $LIB_TAGS
                do
                    if [ -r $t/tags ]
                    then
```

```
                        cat $t/tags >> $TMP_FILE
                    fi
                done;;

        -o)
                others=1;;
    esac

# Append the tags files for any other directories specified.

    if   $others -eq 1 -a $i != "-o" ]

        OTHER_TAGS=$i

        if [ -r "$OTHER_TAGS/tags"
        then
            cat $OTHER_TAGS/tags >> $TMP_FILE
        else
            echo " $OTHER_TAGS/tags not found."


done

# Sort work file and move output to $TAGS_FILE.

sort $TMP_FILE > $TAGS_FILE

# Clean-up working files.

rm $TMP_FILE
exit 0
```

INFORMIX®

Chapter

D

Sample Generic Pick List Window Source Code

© 1993 Informix Software, Inc.

Appendix: Generic Pick List

Generic Pick List Window Module

```
##############################################################
#   Module              :   %M%
#   Description :   Generic pick list window functions
#
#   Function          :     pick_lst
#                           pick_list_show
#                           pick_lst_dsp_ind
#                           sql_open_pick_lst
#                           sql_close_pick_lst
#                           sql_fetch_lst
#   SCCS Id No  : %W%
#   Author              :   <name>
#   Date                :   %H% %T%
#   Path                :   %P%
#   Modification:   %G% %U%
##############################################################

GLOBALS "./global_1.4gl"
```

pick_lst Function

```
#=================================================================
#   Function     :   pick_lst
#   Description :   Serves as the driver routine for displaying a
#                   generic pick list window.
#   Incoming Parameters:sql_stmt -   sql select statement
#                       curr_cd -    holds the value of the
#                                    current code
#                       wnd_dsc -    window description
#                       rw       -   screen row
#                       cl       -   screen column
#   Return Parameters : new_cd   -   selected code
#                       new_dsc -    selected description
#                       val_sel -    flag indicating selection
#                                    of a value
#   Impact Globals      :<none>
#   Perform Screens     :<none>
#
#   Author              :<name>
#   Date                :09/10/93
#   Modification Description: Original Code
#=================================================================

FUNCTION pick_lst ( sql_stmt, curr_cd, wnd_dsc, rw, cl )

    DEFINE
        sql_stmt        CHAR(500),
        curr_cd         CHAR(8),
        wnd_dsc         CHAR(30),
        rw              SMALLINT,
        cl              SMAILLINT,
        sql_stat        INTEGER,
        new_cd          CHAR(8),
        new_dsc         CHAR(30),
        val_sel         SMALLINT

    WHENEVER ERROR CALL err_func

    LET  new_cd = NULL
    LET new_dsc= FALSE
    LET val_sel= FALSE
    LET wnd_dsc= UPSHIFT (wnd_dsc)

    CALL sql_open_pick_lst ( sql_stmt ) RETURNING sql_stat

    IF ( sql_stat = 0) THEN
        LET wnd_dsc = UPSHIFT ( wnd_dsc )
```

Sample Generic Pick List Window Source Code D - 315

```
        CALL pick_lst_show ( wnd_dsc, curr_cd, rw, cl )
            RETURNING new_cd, new_dsc, val_sel
    ELSE
        CALL err_discover ( "sql_open_pick_lst()", G_SQL_ERR,
                            sql_stat, G_NON_FATAL_ERR )
    END IF

    RETURN new_cd, new_dsc, val_sel

END FUNCTION
```

Sample Generic Pick List Window Source Code D - 316

pick_lst_show Function

```
###############################################################
#   Function    :   pick_lst_show
#   Description :   Display the pick list window
#   Incoming Parameters:curr_cd -   holds the value of the
#                                   current code
#                       wnd_dsc -   window description
#                       rw      -   screen row
#                       cl      -   screen column
#   Return Parameters : new_cd  -   selected code
#                       new_dsc -   selected description
#                       val_sel -   flag indicating selection
#                                   of a value
#   Impact Globals  :   <none>
#   Perform Screens :   pick_w.frm
#
#   Author          :   <name>
#   Date            :   09/10/93
#   Modification Description: Original Code
###############################################################

FUNCTION pick_lst_show ( wnd_dsc, curr_cd, rw, cl )

    DEFINE
        wnd_dsc         CHAR(30),
        curr_cd         CHAR(8),
        rw              SMALLINT,
        cl              SMALLINT,

        pa_pick_lst     ARRAY[75] OF RECORD
                            cd          CHAR(8),
                            dsc         CHAR(30)
                        END RECORD

        arr_cnt         SMALLINT,
        sql_stat        INTEGER,
        new_cd          CHAR(8),
        new_dsc         CHAR(30),
        val_sel         SMALLINT

    WHENEVER ERROR CALL err_func

    LET  val_sel    = TRUE
    LET  int_flag   = FALSE
    LET new_cd      = NULL
    LET new_dsc     = NULL
    LET  arr_cnt    = 1
    LET sql_stat= 0
```

```
# Retrieve rows from the open cursor

    WHILE ( sql_stat = 0 ) AND ( arr_cnt != 75 )
        CALL sql_fetch_pick_lst() RETURNING sql_stat,
                                    pa_pick_lst[arr_cnt].*

        IF ( arr_cnt = 75 ) THEN
            CALL error_func("exceed msg")
        ELSE
            IF ( sql_stat = 0 ) THEN
                LET arr_cnt = arr_cnt + 1
            END IF
        END IF
    END WHILE

    CALL sql_close_pick_lst( )

    LET arr_cnt = arr_cnt - 1
# Display an error message if no rows are found, otherwise
# display the pick-list window

    IF ( arr_cnt = 0 ) THEN
        CALL err_msg ( 1 )
    ELSE
        OPEN WINDOW pick_wnd AT rw, cl, WITH FORM "pick_dw"
            ATTRIBUTE ( FORM LINE FIRST, BORDER )

        CALL dsp_title ( wnd_dsc, 1, 34, 1 )
        CALL pick_lst_dsp_ind ( arr_cnt, 10 )
        CALL SET_COUNT ( arr_cnt )

        DISPLAY ARRAY pa_pick_lst TO sad_pck_lst.*

            ON KEY ( CONTROL-P )
                CALL print_screen()
            ON KEY ( ESCAPE, INTERRUPT )
                LET int_flag = TRUE
                EXIT DISPLAY
            ON KEY ( ACCEPT, RETURN )
                EXIT DISPLAY

        END DISPLAY

        CLOSE WINDOW pick_wnd
    END IF

    IF ( int_flag OR arr_cnt = 0 ) THEN
        LET int_flag = FALSE
```

Sample Generic Pick List Window Source Code D - 318

```
        LET val_sel             FALSE

        LET new_cd              curr_cd
        LET new_dsc             NULL

        LET curr_cnt    ARR_CURR()
        LET new_cd              pa_pick_lst[arr_cnt].cd
        LET new_dsc             pa_pick_lst[arr_cnt].dsc
    END IF

    RETURN new_cd, new_dsc, val_sel

END FUNCTION
```

pick_lst_dsp_ind Function

```
##################################################################
#   Function      : pick_lst_dsp_ind
#   Description : Display an indicator to signal if there are more
#                 rows that are not displayed on the first screen
#   Incoming Parameters:arr_cnt -   the number of rows selected
#                       scr_cnt -   the number of rows within
#                                   one screen
#   Return Parameters   : <none>
#   Impact Globals      : <none>
#   Perform Screens     : <none>
#
#   Author              : <name>
#   Date                : 09/10/93
#   Modification Description: Original Code
##################################################################

FUNCTION pick_lst_dsp_ind ( arr_cnt, scr_cnt )

    DEFINE
        arr_cnt             SMALLINT,
        scr_cnt             SMALLINT

    WHENEVER ERROR CALL err_func

# Display a message to indicate if more rows exist within the
# picklist

    IF ( arr_cnt <= scr_cnt ) THEN
        DISPLAY "( end )" TO MORE
    ELSE
        DISPLAY "( more )" TO MORE
    END IF

END FUNCTION
```

Sample Generic Pick List Window Source Code D - 320

sql_open_pick_lst Function

```
##################################################################
#   Function     : sql_open_pick_lst
#   Description : PREPARE's and DECLARE's the pick list SQL
#                 statement
#   Incoming Parameters:sql_stmt -  SQL pick list statement
#   Return Parameters  :sql_stat -  SQL statement status
#                       arr_cnt  -  number of rows processed
#                       max_exceed- maximum array size exceeded
#                                   flag
#   Impact Globals     :ga_pick_lst-holds the pick list
#                                   information
#   Perform Screens    :<none>
#
#   Author             :<name>
#   Date               : 09/10/93
#   Modification Description: Original Code
##################################################################

FUNCTION sql_open_pick_lst (sql_stmt)

    DEFINE
        sql_stmt            CHAR(512),
        sql_stat            SMALLINT

    WHENEVER ERROR CONTINUE

    PREPARE pick_lst_prep FROM sql_stmt

    LET sql_stat = SQLCA.SQLCODE

    IF ( sql_stat = 0 ) THEN
        DECLARE pick_lst_curs CURSOR FOR pick_lst_prep
        OPEN pick_lst_curs

        LET sql_stat = SQLCA.SQLCODE
    END IF

    WHENEVER ERROR CALL err_func

    RETURN sql_stat

END FUNCTION
```

Sample Generic Pick List Window Source Code D - 321

sql_close_pick_lst Function

```
##############################################################
#   Function    : sql_close_pick_lst
#   Description : Close and free the pick list cursor
#   Incoming Parameters: <none>
#   Return Parameters  : <none>
#   Impact Globals     : <none>
#   Perform Screens    : <none>
#
#   Author             : <name>
#   Date               : 09/10/93
#   Modification Description: Original Code
##############################################################

FUNCTION sql_close_pick_lst()

    WHENEVER ERROR CONTINUE

    CLOSE pick_lst_curs
    FREE pick_lst_prep

    WHENEVER ERROR CALL err_func

END FUNCTION
```

sql_fetch_pick_lst Function

```
############################################################
#   Function    : sql_fetch_pick_lst
#   Description : FETCH's the pick list information from the
#                 cursor OPENed in sql_open_pick_lst() function
#   Incoming Parameters     : <none>
#   Return Parameters       : sql_stat  - SQL statement status
#                             cd        - pick list code
#                             dsc       - pick list description
#   Impact Globals          : <none>
#   Perform Screens         : <none>
#
#   Author                  : <name>
#   Date                    : 09/10/93
#   Modification Description: Original Code
############################################################

FUNCTION sql_fetch_pick_lst()

    DEFINE
        sql_stat            SMALLINT,
        cd                  CHAR(12),
        dsc                 CHAR(30)

    LET cd      = NULL
    LET dsc=     NULL

    WHENEVER ERROR CONTINUE

    FETCH pick_lst_curs INTO cd, dsc

    LET sql_stat = SQLCA.SQLCODE

    IF ( sql_stat != 0 ) THEN
        CALL sql_close_pick_lst()
    END IF

    WHENEVER ERROR CALL error_func

    RETURN sql_stat, cd, dsc

END FUNCTION
```